Abstract

Modular Machine Code Verification

Zhaozhong Ni

2007

Formally establishing safety properties of software presents a grand challenge to the computer science community. Producing proof-carrying code, *i.e.*, machine code with machine-checkable specifications and proofs, is particularly difficult for system softwares written in low-level languages. One central problem is the lack of verification theories that can handle the expressive power of low-level code in a modular fashion. In particular, traditional type- and logic-based verification approaches have restrictions on either expressive power or modularity.

This dissertation presents XCAP, a logic-based proof-carrying code framework for modular machine code verification. In XCAP, program specifications are written as general logic predicates, in which syntactic constructs are used to modularly specify some crucial higher-order programming concepts for system code, including embedded code pointers, impredicative polymorphisms, recursive invariants, and general references, all in a logical setting. Thus, XCAP achieves the expressive power of logic-based approaches and the modularity of type-based approaches. Its meta theory has been completely mechanized and proved.

XCAP can be used to directly certify system kernel code. This dissertation contains a mini certified thread library written in x86 assembly. Every single instruction in the library, including those for context switching and thread scheduling, has a formal XCAP specification and a proof. XCAP is also connected to existing certifying compiler; a type-preserving translation from a typed assembly language to XCAP is included.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE
MAY 2007

2. REPORT TYPE

3. DATES COVERED
00-00-2007 to 00-00-2007

4. TITLE AND SUBTITLE
Modular Machine Code Verification

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Yale University ,Department of Computer Science,New Haven,CT,06520-8285

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Formally establishing safety properties of software presents a grand challenge to the computer science community. Producing proof-carrying code, i.e., machine code with machine-checkable specifications and proofs, is particularly difficult for system softwares written in low-level languages. One central problem is the lack of verification theories that can handle the expressive power of low-level code in a modular fashion. In particular, traditional type- and logic-based verification approaches have restrictions on either expressive power or modularity. This dissertation presents XCAP, a logic-based proof-carrying code framework for modular machine code verification. In XCAP, program specifications are written as general logic predicates, in which syntactic constructs are used to modularly specify some crucial higher-order programming concepts for system code, including embedded code pointers, impredicative polymorphisms, recursive invariants, and general references, all in a logical setting. Thus, XCAP achieves the expressive power of logic-based approaches and the modularity of type-based approaches. Its meta theory has been completely mechanized and proved. XCAP can be used to directly certify system kernel code. This dissertation contains a mini certified thread library written in x86 assembly. Every single instruction in the library, including those for context switching and thread scheduling, has a formal XCAP specification and a proof. XCAP is also connected to existing certifying compiler; a typepreserving translation from a typed assembly language to XCAP is included.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**183**	

Modular Machine Code Verification

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

By
Zhaozhong Ni

Dissertation Director: Zhong Shao

May 2007

Copyright © 2007 by Zhaozhong Ni

All rights reserved.

Contents

List of Figures

Acknowledgments

The first person I want to thank is my advisor, Professor Zhong Shao. Entering the PhD program six years ago, I had little idea about what it would actually take to finish this dissertation. It is Professor Shao's encouraging advices and thoughtful guidance that helped and supported my pursuit of this exciting research. He patiently taught me knowledge and trained my skills starting with the very basics. From day one, he has been amazingly good at pushing me towards working with him more as a colleague than as a student. I remember and appreciate all the lively discussions, exciting discoveries, as well as disappointing failures and sometimes heated debates I shared with him during the past six years. I truly learned a lot from his broad vision on research, persistency in deep understanding, and ever-lasting enthusiasm.

I have been very lucky to have overlap with every other member of the FLINT group in its first ten years. Valery Trifonov was extremely helpful to my PhD study, especially during the first few years. Bratin Saha, Christopher League, and Stefan Monnier answered many of my naïve questions in the early years. I had close and pleasant collaborations with Dachuan Yu, Nadeem A. Hamid, and Xinyu Feng. Entering the PhD program in the same year, Hai Fang and I discussed and exchanged our experiences and frustrations during the various stages of the study. I also enjoyed the discussions with Andrew McCreight, Rodrigo Ferreira, and Alexander Vaynberg.

[0] This research is based on work supported in part by gifts from Intel and Microsoft, DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, NSF ITR grant CCR-0081590, and NSF grant CCR-0524545. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

I would like to thank Professor Paul Hudak, Professor Carsten Schürmann, and Professor David Walker for serving on my thesis committee and being my thesis readers. Professor Paul Hudak taught me a formal semantics course and has been very helpful to my study in the PhD program. Professor Carsten Schürmann taught me my first formal method course and gave me many advices, especially in my first year. Professor David Walker gave me many instructional advices and had very helpful interactions with me on my research progress.

I would also like to thank Professor Drew McDermott and Professor Yang R. Yang for their help in my graduate study. I want to thank Linda Dobb and Judy Smith, who helped me with all the administrative stuffs.

Finally, I would like to thank my parents and my family, without the support from which the completion of my graduate study and this dissertation would be impossible. I am especially grateful to my wife, Zhiyan, whose love, care, and support are my source of energy along the journey.

Chapter 1

Introduction

We begin this dissertation with the following code:

```
swapcontext:
    ; save old context              ; load new context
    mov eax, [esp+4]                mov eax, [esp+8]
    mov [eax+_eax], 0               mov esp, [eax+_esp]
    mov [eax+_ebx], ebx             mov ebp, [eax+_ebp]
    mov [eax+_ecx], ecx             mov edi, [eax+_edi]
    mov [eax+_edx], edx             mov esi, [eax+_esi]
    mov [eax+_esi], esi             mov edx, [eax+_edx]
    mov [eax+_edi], edi             mov ecx, [eax+_ecx]
    mov [eax+_ebp], ebp             mov ebx, [eax+_ebx]
    mov [eax+_esp], esp             mov eax, [eax+_eax]
                                    ret
```

This 19 lines of x86 assembly code constitutes a common routine for machine-context switching, omitting floating-point and special registers. The first (left) half of the code saves the current machine context, whereas the second (right) half loads the new machine context and resumes execution from there. Being executed virtually every millisecond on most PCs, code sequences similar to this are a crucial part of modern OS.

Surprisingly, despite the simplicity and small size of this piece of code, to the author's knowledge, it has never been rigorous proved to be safe and correct, whether manually or automatically. There is not even a rigorous statement on what this 19 lines of code should and should not do. In fact, this seemingly simple code sequence turns out to be extremely difficult to specify and reason about. The lack of safety guarantee at such a core of OS kernel makes any claim about software safety and security a "wish" instead of reality.

1.1 Machine Code Verification

Program correctness or, to a less degree, code safety, has long been one of the most desired goals for computer programmers and users. With great dependency on information technology in the modern world, individuals and businesses are getting more concerned about the reliability and the security of the computing infrastructure than ever. Great efforts and investments have been put into the research and development of software verification tools for a long time. In [32], formally establishing safety properties of software has been identified as a grand challenge to the computer science community.

From the point of view of software engineering, there are many levels of program abstractions. At the top level are algorithms and protocols, which normally get implemented in various high-level programming languages (including domain-specific languages), the later of which often get translated into and are supported by intermediate languages found either in compilers or in portable virtual machines. In the end there is assembly and machine code, which carries out the actual computations and provides critical runtime support.

Surprisingly, in the entire hierarchy of program verification there is one important level that receives much less attention than the others. Despite the fact that all higher level abstraction and verification efforts will eventually be based upon the machine code level, there have been much fewer existing works done in machine code verification. The direct result of the lack of machine code level verification support is that most of the safe algorithms/protocols/programs usually lose their safety guarantees when being implemented or translated into binary.

Recently, formal studies on critical code such as OS kernels are attracting growing interests. Among other techniques, type systems and program logics have been widely applied in modern programming language and OS researches. One representative example is the Singularity project [33], which aims to build a highly reliable OS using a type-safe programming language (C#) and other techniques for program specification and verification. Another example is the Verisoft project [23], which uses computer-aided logical proofs to obtain assurance of the correctness of critical systems including OS kernels. Unfortunately, both fall short at the code for context switching at the beginning of this chapter: Singularity uses unsafe assembly code for programming context switching; Verisoft has not approached below the level of user processes in software verification.

The lack of work on self-contained verification of machine level context and thread implementation reveals subtle limitations of traditional methods. For example, although type systems provide excellent support for modularity and higher-order features, they tend to be very specialized when dealing with low-level system code, raising both flexibility and interoperability issues. As another example, many program logics suffer from the weak support on higher-order features such as embedded code pointers [53], as will be discussed in details in this dissertation.

The purpose of this thesis work is to advance the research of machine code verification. In particular, we want to achieve a better balance between expressiveness and modularity over existing methods for low-level system and application code verification. Although theoretically one could apply any verification methods on machine code, expressiveness and modularity are two extremely important criteria for machine code verification.

One of the main reasons that people write code in low-level languages instead of high-level languages is the expressiveness of the low-level languages themselves. The speed, flexibility, and precise control of machine resources are examples of the expressiveness of machine languages. To verify such kind of code, the expressiveness of the verification system has to be much more powerful than those used for high-level languages.

Separate compilation is important for high-level programs. While machine code does not need to be compiled, in many cases it still needs to be linked together before execution. Moreover, separate verification is even more important for machine code than high-level programs, since machine code is generally larger in size and more tedious than a corresponding high-level program. Its verification condition and process are also generally much larger and slower than the high-level's. For a machine code verification system to be practical and efficient, it must have as good support of modularity as the high-level verification systems.

1.2 Previous Work

In this section, we discuss existing efforts on low-level and machine code verification.

Typed assembly languages. Type system has been popular and proved successful in programming languages. Thus, a natural way to address the machine and assembly level program verification is to design a type system for assembly language. Partly inspired by the Typed Intermediate Language (TIL) [57], Type Assembly Language (TAL) [41] is a first effort to address the low-level code safety problem in the traditional type system manner. The specification of TAL is written in syntactically defined types, including integer, tuple, code pointer, polymorphic, and existential types. Since the type checking of TAL is syntax directed and decidable, the verification process is fully automatic. By using certifying compilation technology, many of the existing high-level programming languages can be translated into TAL code with type annotation. Later improvement of TAL includes stack and exception [40], modularity and dynamic linking [24], dependent type and arrays [60], recursive type and connections with foundational proof-carrying code [29], heterogeneous tuples and disjoint sums [13], low-level optimization [11].

However, there are lot safety policies which have not been supported by the above work on TAL. For example, memory management are not handle in any of them. The fact that most of TAL languages require a hard-wired "malloc" or "alloc" instruction, which is generally a library function above the machine instruction level, is due to the inability of them to type check precise memory management. To support these safety properties in TAL, one would need to incorporate more and more features into the type system and the interoperations between different versions of TALs may also be problematic.

Proof-carrying code. Traditionally, the code consumer receives binary from the code producer and verify the code before executing it. To shift the burden of verification from code consumer to code producer, Proof-Carrying Code (PCC) [45, 43, 44, 12] allows a code producer to provide a machine (DEC Alpha) language program to a host along with a formal proof of its safety. The specification language in PCC is typing predicates with some universal logic connectors and the safety policy is expressed as pre/post conditions of code sequences. The proofs are written in a logic extended with many language-specific typing rules. By using verification-condition generator (VCgen), both the code producer and the code consumer can automatically calculate the safety requirements (lemmas) at each instruction from the safety policy. For those simple examples, the theorem prover can automatically prove those lemmas. Code consumer would simply use a proof checker to ensure that the proofs do prove those lemmas. By proving the soundness of the VCgen algorithm, the code consumer can be sure that the safety policy has been satisfied. Since the source language's type system is somehow embedded in the logic, PCC can enjoy most of the benefits of source type systems such as modularity and support more general properties through logic formulas.

The VCgen and proof checker are inside the trusted computing base and are error-prone since they all involve language-specific typing rules which themselves are not always error-free. For example, [36] discovered a serious bug in the typing rules. Another difficulty of PCC is that language-specific typing rules makes interoperation between code written in different languages ad hoc, if not hard at all.

Foundational proof-carrying code. Foundational Proof-Carrying Code (FPCC) [6, 5] tackles the problems of PCC by constructing and verifying its proofs using strictly the foundations of mathematical logic, with no type-specific axioms. The machine semantics of Sun Sparc is formalized in the logic in the first place. VCgen can now be removed from trusted computing base. Instead of directly verifying binaries in the machine level, exiting work of FPCC all takes programs written in a slightly higher-level TAL-like language and automatically general their corresponding FPCC according to the types and typing derivations. In the case of semantic approach [6, 18, 3, 8] types, typing derivations, and soundness proof are directly interpreted into foundational logic's formula and proofs. The syntactic approach [29] simply encodes the syntactic type system in the logic and obtain the FPCC by mapping between machine and FTAL steps and utilizing the encoded FTAL soundness proof.

Automated proofs of object code for a widely used microprocessor. Yu [65] uses a limited first-order predicate logic (ACL) to define and specify the machine (Motorola MC68020) and safety policy (Hoare-style pre/post-condition relation on binary encoding of machine program segment). The proof of the safety properties is done by manually specifying lemmas needed and their usage, and letting the Boyer-Moore Theorem Prover (Nythm) to complete the whole proof. There the safety properties are mostly correctness of functions, including those in a unix string library.

Cyclone / C-Cured / Vault. Cyclone [35], CCured [46], and Vault [16] are safe C-like languages which aim at providing verification support at a level close to assembly. They could serve as the low-level system programming language connecting with machine code verification system when a very expressive specification language is not required.

JVML / MSIL. Java Virtual Machine Language [27] and Microsoft Intermediate Language [37] are intermediate-level portable languages for virtual machines. All Java and C# programs are compiled to and distributed in these languages. These languages can be further compiled to TAL-like languages.

1.3 Challenges and Contributions

The work in this dissertation lies within the PCC (and FPCC) framework. In principle, PCC can be used to verify safety properties of arbitrary machine-language programs.

Existing PCC systems [12, 29, 11, 13], however, have focused on programs written in type-safe languages [26, 38] or variants of typed assembly languages [41]. Type-based approaches are attractive because they facilitate automatic generation of the safety proofs (by using certifying compilers) and provide great support to modularity and higher-order language features. But they also suffer from several serious limitations. First, types are not expressive enough to specify sophisticated invariants commonly seen in the verification of low-level system software. Recent works on *logic-based type systems* [61, 55, 14, 1, 4] have made types more expressive but they still cannot specify advanced state invariants and assertions [54, 63, 64] definable in a general-purpose predicate logic with inductive definitions [58]. Second, type systems are too weak to prove advanced properties and program correctness, especially in the context of concurrent assembly code [63, 20]. Finally, languages of different style often require different type systems, making it hard to reason about interoperability.

An alternative to type-based methods is to use Hoare logic [22, 31]—a widely applied technique in program verification. Hoare logic supports formal reasoning using very expressive assertions and inference rules from a general-purpose logic. In the context of foundational proof-carrying code (FPCC) [5, 63], the assertion language is often unified with the mechanized meta logic (following Gordon [25])—proofs for Hoare logic consequence relation and Hoare-style program derivations are explicitly written out in a proof assistant. For example, Touchstone PCC [44] used Hoare-style assertions to express complex program invariants. Appel *et al* [6, 9] used Hoare-style state predicates to construct a general semantic model for machine-level programs. Yu *et al* [63, 64, 20] recently developed a certified assembly programming framework that uses Hoare-style reasoning to verify low-level system libraries and general multi-threaded assembly programs.

Unfortunately, Hoare logic, as Reynolds [54] observed, does not support higher-order features such as embedded code pointers (ECPs) well. Similarly, features such as impredicative polymorphisms, recursive specifications, and weak update references, are not handled well simultaneously in the logical setting. How to modularly support these features without sacrificing the expressive power of logic is the first and major challenge we face when doing machine code verification.

The second challenge we face when doing machine code verification is whether the verification method is applicable to real-world program verification on realistic system kernel code.

The third challenge is whether the results of verification is an isolated one or can actually connect with other verification systems such as type systems.

The contributions of this dissertation can be summarized into the following aspects, which corresponds to the structure of the dissertation.

Embedded code pointer and the basic XCAP framework. We designed the XCAP framework, which provides the first simple and general solution to the ECP problem for Hoare-logic verification systems. By "simple", we mean that our method does not alter the structure of Hoare-style program derivations and assertions. By "general", we mean that our technique can indeed handle all kinds of machine-level ECPs, including those hidden inside higher-order closures. Relevant details can be found in Chapter 3.

Impredicative polymorphisms and recursive specifications. We extended the XCAP framework to support impredicative polymorphisms and recursive specifications. Both extensions cause very little change in the XCAP inference rules and meta theory. Thus, they are light weight and proved sound. Using the extended XCAP, we solve the ECP problem for separation logic [54] and present a verification of a destructive list-append function, which is listed as an example in [54]. Relevant details can be found in Chapter 4.

Weak update and a translation from typed assembly language. Weak update, also termed as "general reference", is another higher-order feature that logic-based verification methods fail to support well. We once again extended the XCAP framework to support weak update, using similar syntactic technique for the support of ECP in Chapter 3.

We then explore the relationship between XCAP and typed assembly languages (TAL). TAL and XCAP are suitable for different kinds of verification tasks. Previously, programs verified in either one of them can not interoperate freely with the other, making it hard to integrate them into a complete system. Moreover, the relationship between TAL and CAP/XCAP lines of work has not been discussed extensively.

We compare the type-based and logic-based methods by presenting a type-preserving translation from a TAL language to XCAP. The translation involves an intermediate step of a "semantic" TAL language. Our translation supports polymorphic code, mutable reference, existential, and recursive types. Since we proved typing preservation for the translation from TAL to XCAP, there is a clear path to link and interoperate well-typed programs from traditional certifying compilers with certified libraries by XCAP. Relevent details can be found in Chapter 5.

A port of XCAP to the x86 architecture. We port XCAP to a faithful subset of the x86 architecture, which adds the support of instruction decoding, finite machine word, word-aligned byte-addressed memory, conditional flags, built-in stack and push/pop instructions, and function call/return instructions. On top of it, we made practical adaptations and built useful abstractions, particularly on the handling of the stack and function calls. We demonstrate its usage and these abstractions for the verification of a polymorphic queue module. Relevent details can be found in Chapter 6.

A certified mini thread library. We mechanically verified a thread implementation at the machine level using the ported XCAP. Using the first mechanized proof for the safety of a machine-level thread implementation, we demonstrate the power and practicality of the XCAP framework, and thus the fact that the certification of complex machine-level system code is not beyond reach. The specifications and proof of MTH modules and routines are modular. Each piece of the code is specified and proved with minimal reference to external code. For example, in the verification of context module, there is no mention of thread at all. More details can be found in Chapter 7.

Mechanization. One key feature of the XCAP framework is mechanization. Not only did we mechanize our machine syntax, machine semantics, assertion languages, interpretations, inference rules, and program specifications and proofs in a general mathematic logic, we also mechanized the complete meta theory of XCAP. Plus, we want to directly obtain the power of higher-order predicate logic, through a shallow embedding. For these reasons, our usage of the Coq proof assistant [58] is to treat it as both a mechanized logic framework and a mechanized meta logic framework. More details can be found in Chapter 8.

Chapter 2

Background

Most of the theoretical presentations and discussions in this dissertation are based on a same RISC-like ideal target machine (TM). Most of the verification systems and proofs in this dissertation assume a same underlying formal meta logic—a variant of the Calculus of Inductive Constructions (CiC) [50], upon which the Coq proof assistant is based. For proof mechanization purpose, we refer to it as our "mechanized meta logic". Certified Assembly Programming (CAP) is the logic-based machine code verification framework from which the work in this dissertation evolves. In this chapter, we prepare the reader with the machine, the logic, and the baseline CAP.

2.1 The Target Machine

All theoretical results presented in this dissertation share a common raw target machine TM, as defined in Figure 2.1. A TM program ($\mathbb{P}$), an entire machine configuration, consists of a code heap ($\mathbb{C}$), a dynamic state component ($\mathbb{S}$) made up of a register file ($\mathbb{R}$) and a data heap ($\mathbb{H}$), and an instruction sequence ($\mathbb{I}$) to be executed next. Code heap is a collection of code labels ($\mathtt{f}$) and the code sequence they points to. The register file is made up of 32 registers, while the data heap is a partial mapping from data labels ($\mathtt{l}$) to machine words ($\mathtt{w}$). In essence, $\mathtt{f}$, $\mathtt{l}$, and $\mathtt{w}$ are all just plain natural numbers (i). Since TM has an infinite

$$
\begin{array}{rlll}
(\mathit{Program}) & \mathbb{P} & ::= & (\mathbb{C},\mathbb{S},\mathbb{I}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \leadsto \mathbb{I}\}^{*} \\
(\mathit{State}) & \mathbb{S} & ::= & (\mathbb{H},\mathbb{R}) \\
(\mathit{Mem}) & \mathbb{H} & ::= & \{\mathtt{l} \leadsto \mathtt{w}\}^{*} \\
(\mathit{Regfile}) & \mathbb{R} & ::= & \{\mathtt{r} \leadsto \mathtt{w}\}^{*} \\
(\mathit{Reg}) & \mathtt{r} & ::= & \{\mathtt{r}_k\}^{k\in\{0\ldots31\}} \\
(\mathit{Word},\mathit{Labels}) & \mathtt{w},\mathtt{f},\mathtt{l} & ::= & i \;\; (\mathit{nat\ nums}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= & \mathtt{c};\mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{r} \\
(\mathit{Instr}) & \mathtt{c} & ::= & \mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s\mathtt{r}_t \mid \mathsf{addi}\ \mathtt{r}_d,\mathtt{r}_s,i \mid \mathsf{alloc}\ \mathtt{r}_d,i \mid \mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f} \\
 & & & \mid \mathsf{free}\ \mathtt{r}_s,i \mid \mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(i) \mid \mathsf{mov}\ \mathtt{r}_d,\mathtt{r}_s \mid \mathsf{movi}\ \mathtt{r}_d,i \mid \mathsf{st}\ \mathtt{r}_d(i),\mathtt{r}_s
\end{array}
$$

Figure 2.1: Syntax of target machine

word size, we expects its data heap to be bounded but infinite—there is no upper limit on how large a data heap can be, but at any program execution point, all the allocated heap cells are below a certain boundary (this is because in TM one can only allocate a finite amount of memory in each allocation).

TM follows the RISC style for simplicity. Its instruction set of TM is minimal but extensions are straightforward. There are instructions ($\mathtt{c}$) for arithmetic operations (add $\mathtt{r}_d,\mathtt{r}_s\mathtt{r}_t$, addi $\mathtt{r}_d,\mathtt{r}_s,i$), data movements (mov $\mathtt{r}_d,\mathtt{r}_s$ and movi $\mathtt{r}_d,i$), memory allocation/deallocation (alloc $\mathtt{r}_d,i$ and free $\mathtt{r}_s,i$), memory accesses (ld $\mathtt{r}_d,\mathtt{r}_s(i)$ and st $\mathtt{r}_d(i),\mathtt{r}_s$), and condition jumps (bgti $\mathtt{r}_s,i,\mathtt{f}$). Each code block (instruction sequence) must end with either a direct jump (jd $\mathtt{f}$) or an indirect jump (jmp $\mathtt{r}$).

The operational semantics of this language (see Figure 2.2) should pose no surprise. The add instructions add values from source registers and immediate values and put the sums into the destination registers. The data movement instructions take either the source register value or an immediate value, and put it into the destination register. Heap allocation instruction takes a required size i and allocates a continuous memory block of that size, with its initial value undefined (so for safe execution of programs, these values should not be used). To dispose a piece of memory, every cell in it must be previously allocated. Similarly, to access a memory cell, it must be previously allocated. Whether

if $\mathbb{I} =$	then $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}) \longmapsto$
jd $\mathtt{f}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathtt{f}))$ when $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$
jmp $\mathtt{r}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathbb{R}(\mathtt{r})))$ when $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\mathbb{C})$
bgti $\mathtt{r}_s, i, \mathtt{f}; \mathbb{I}'$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \leq i$; $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathtt{f}))$ when $\mathbb{R}(\mathtt{r}_s) > i$
$\mathtt{c}; \mathbb{I}'$	$(\mathbb{C},\mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R}),\mathbb{I}')$

where

if $\mathtt{c} =$	then $\mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R}) =$
add $\mathtt{r}_d, \mathtt{r}_s \mathtt{r}_t$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\})$
addi $\mathtt{r}_d, \mathtt{r}_s, i$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)+i\})$
mov $\mathtt{r}_d, \mathtt{r}_s$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)\})$
movi $\mathtt{r}_d, i$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto i\})$
alloc $\mathtt{r}_d, i$	$(\mathbb{H}\{\mathtt{l} \leadsto \_, \ldots, \mathtt{l}+i-1 \leadsto \_\},\mathbb{R}\{\mathtt{r}_d \leadsto \mathtt{l}\})$ where $\mathtt{l}, \ldots, \mathtt{l}+i-1 \notin \mathsf{dom}(\mathbb{H})$ and $\_$ is a random value
free $\mathtt{r}_s, i$	$(\mathbb{H}/\{\mathbb{R}(\mathtt{r}_s), \ldots, \mathbb{R}(\mathtt{r}_s)+i-1\},\mathbb{R})$ when $\mathbb{R}(\mathtt{r}_s), \ldots, \mathbb{R}(\mathtt{r}_s)+i-1 \in \mathsf{dom}(\mathbb{H})$
ld $\mathtt{r}_d, \mathtt{r}_s(i)$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+i)\})$ when $\mathbb{R}(\mathtt{r}_s)+i \in \mathsf{dom}(\mathbb{H})$
st $\mathtt{r}_d(i), \mathtt{r}_s$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_d)+i \leadsto \mathbb{R}(\mathtt{r}_s)\},\mathbb{R})$ when $\mathbb{R}(\mathtt{r}_d)+i \in \mathsf{dom}(\mathbb{H})$

Figure 2.2: Operational semantics of target machine

directly jumping to an immediate code label or a register value, the target address must point to a piece of code in the code heap. Branch instruction compares the register value with an immediate value, and decides if it should continue with the next instruction or jump to somewhere else. In all the above cases, when the side-conditions of instructions are unsatisfied, the execution of TM gets "stuck." To guarantee that a TM program will never enter the "stuck" status is the minimum requirement for any verification systems.

2.2 Mechanized Meta Logic

Both the syntax and the operational semantics of TM, as well as all the verifications systems, program specifications, and proofs in this dissertation, are defined upon a formal meta logic. In this dissertation we use a variant of the calculus of inductive constructions (CiC) [50], which is a higher-order predicate logic extended with powerful inductive def-

$$(Term)\ A,B ::= Set \mid Prop \mid Type \mid x \mid \lambda x\!:\!A.\,B \mid A\ B \mid A \rightarrow B \mid \Pi x\!:\!A.\,B \mid \textit{inductive definitions}$$
$$(Prop)\ p,q ::= \mathsf{True} \mid \mathsf{False} \mid \neg p \mid p \wedge q \mid p \vee q \mid p \supset q \mid \forall x\!:\!A.\,p \mid \exists x\!:\!A.\,p \mid \ldots$$

Figure 2.3: Mechanized meta logic

initions. We informally present our formal meta logic in Figure 2.3. Terms in CiC can be sorts (Set, Prop, and Type), variables, function abstractions, function applications, non-dependent products, dependent products, or inductive definitions. We omit the details of inductive definitions and will instead use examples to explain them later. The logic part of CiC contains terms of Prop sort, which provides common logic quantifiers and connectives, and allows user expansions by inductively defined propositions.

Since one goal of the work in this dissertation is to generate machine-checkable proofs, we want our formal meta logic to be a mechanized one. We select CiC based on the fact that it is the underlying logic theory for the Coq proof assistant [58]. We use Coq for two purposes. The first is to use it as a mechanized meta logic framework, in which we can represent all the syntax, rules, and meta theory of our machine and verification systems in the *Set* and *Type* sort of Coq. For example, to represent TM, machine state can be embedded as a *State* type (which has *Set* sort in Coq); registers, machine instruction sequences, commands, and execution semantics can be defined as inductive definitions.

```
Inductive Register : Set := r0 | r1 | ... | r31.

Inductive Command  : Set := add  : Register -> Register -> Register -> Command
                          | addi : Register -> Register -> Word     -> Command
                          | mov  : Register -> Register             -> Command
                          | ... .

Inductive InstrSeq : Set := iseq : Command -> InstrSeq -> InstrSeq
                          | jd   : Word                -> InstrSeq
                          | jmp  : Register            -> InstrSeq
                          | ... .

Inductive Next : Command -> State -> State -> Prop :=
  | stp_add : forall rd rs rt H R,
                Next (add rd rs rt) (H, R) (H, uR R rd (R rs + R rt))
  | stp_mov : forall rd rs H R,
                Next (mov rd rs) (H, R) (H, uR R rd (R rs))
  | stp_ld  : forall rd rs w H R w’,
                lookup H (R rs + w) w’ ->
                Next (ld rd rs w) (H, R) (H, uR R rd w’)
  | ... .

Inductive STEP : Program -> Program -> Prop :=
  | stp_iseq : forall C S S’ c I,
                 Next c S S’ ->
                 STEP (C, (S, iseq c I)) (C, (S’, I))
  | stp_jd   : forall C S l I,
                 lookup C l I ->
                 STEP (C, (S, jd l)) (C, (S, I))
  | stp_jmp  : forall C S r I,
                 lookup C (_R S r) I ->
                 STEP (C, (S, jmp r)) (C, (S, I))
  | ... .
```

General safety policies can then be defined as meta logic predicates over the entire machine configuration; they will have the $Program \to Prop$ type. For example, the simple "non-stuckness" safety policy for TM can be defined as follows:

$$\lambda\mathbb{P}.\forall n{:}Nat.\ \exists\mathbb{P}'{:}Program.\ \mathbb{P}\longmapsto^{n}\mathbb{P}'.$$

Here $\longmapsto^{n}$ is the composition of "$\longmapsto$" for n times. Under this setting, if $Safe()$ denotes a particular customized safety policy, a certified binary is just a pair of program $\mathbb{P}$ together with a proof object of type $Safe(\mathbb{P})$, all represented in the mechanized meta logic.

The second usage of CiC/Coq is to directly use its build-in higher-order predicate logic (*Prop*) to serve as (part of) the assertion logics, which are used to write program specifications. Thus there is no need to define a complete new general logic and its meta theory, which makes the approaches in this paper lightweight. In other word, we use shallow embedding instead of deep embedding for the higher-order predicate logic part of the assertion languages in this dissertation. Actually, right in the next section, we will see how the original CAP utilizes this technology and becomes an extremely simple framework.

It is important to note that, although the theory and implementation in this dissertation are presented and done with CiC/Coq, we believe that the key ideas from our results are applicable to other mechanized meta logics.

2.3 The Certified Assembly Programming Framework

As suggested by its name, certified assembly programming, CAP [63] is a logic-based verification framework for machine code verification. In essence, CAP is a Hoare-logic framework for reasoning about assembly programs. Note that the CAP framework presented in this section is slightly different from the original one, as we want to let it share a very similar structure with the work in this dissertation.

Traditionally, Hoare-logic systems usually use the following judgment to reason about program safety and correctness.

$$\{precondition\}\ statement\ \{postcondition\}$$

Where pre- and post-conditions are written in certain assertion logic, and describe the state before and after the execution of *statement*.

In this dissertation, however, TM programs are often written in continuation-passing style [7], as there are no instructions directly in correspondence with function call and return in a high-level language. Hence post-conditions in Hoare logic do not have explicit counterparts in CAP; they are often interpreted as preconditions of the return continuations. The basic form of judgment in CAP is

$$\{\mathtt{a}\}\ \mathbb{I}$$

$$\begin{array}{rccl}
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \leadsto \mathtt{a}\}^* \\
(\mathit{Assertion}) & \mathtt{a} & \in & \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{AssertImp}) & \mathtt{a} \Rightarrow \mathtt{a}' & \triangleq & \forall \mathbb{S}.\, \mathtt{a}\, \mathbb{S} \supset \mathtt{a}'\, \mathbb{S} \\
(\mathit{StepImp}) & \mathtt{a} \Rightarrow_{\mathtt{C}} \mathtt{a}' & \triangleq & \forall \mathbb{S}.\, \mathtt{a}\, \mathbb{S} \supset \mathtt{a}'\, \mathtt{Next}_{\mathtt{C}}(\mathbb{S})
\end{array}$$

Figure 2.4: Assertion language of CAP

where $\mathbb{I}$ is the code block to be reasoned about, and $\mathtt{a}$ is an assertion describing the expectation on machine states before executing $\mathbb{I}$.

Assertion language. Following Gordon [25], CAP's assertion language, as presented in Figure 2.4, is directly unified with the underlying mechanized meta logic (*i.e.*, shallow embedding). CAP assertions ($\mathtt{a}$) tracks the state component in the machine configuration, so any terms of type *State* → *Prop* are valid CAP assertions. For example, an assertion specifying that the registers $\mathtt{r}_1$ and $\mathtt{r}_2$ store a same value and the register $\mathtt{r}_3$ contains a non-NULL value can be written as:

$$\lambda(\mathbb{H},\mathbb{R}).\, \mathbb{R}(\mathtt{r}_1) = \mathbb{R}(\mathtt{r}_2) \wedge \mathbb{R}(\mathtt{r}_3) \neq \mathsf{NULL}.$$

To simplify the presentation, we lift propositional implication ($\supset$) to assertion level ($\Rightarrow$).

In Figure 2.4, there is a construct named code heap specification (Ψ) for expressing user-defined safety requirements on program. A code heap specification associates every code label with an assertion, with the intention that the pre-condition of a code block is described by the corresponding assertion.

Inference rules. CAP defines a set of inference rules for proving judgments for well-formed programs, code heaps, and instruction sequences (see Figure 2.5).

Top-down, a TM program is well-formed (rule PROG) under assertion $\mathtt{a}$ if both the global code heap and the current instruction sequence are well-formed and the machine state satisfies assertion $\mathtt{a}$.

To support separate verification of code modules, we have made some changes to

$\boxed{\Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{P}}$ **(*Well-formed Program*)**

$$\frac{\Psi_{G} \vdash \mathbb{C} : \Psi_{G} \qquad (\mathtt{a}\;\mathbb{S}) \qquad \Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi_{G} \vdash \{\mathtt{a}\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})}\ (\textsc{prog})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C} : \Psi}$ **(*Well-formed Code Heap*)**

$$\frac{\Psi_{IN} \vdash \{\mathtt{a}_i\}\,\mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \rightsquigarrow \mathbb{I}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathbb{I}_n\} : \{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\}}\ (\textsc{cdhp})$$

$$\frac{\begin{array}{c}\Psi_{IN1} \vdash \mathbb{C}_1 : \Psi_1 \qquad \Psi_{IN2} \vdash \mathbb{C}_2 : \Psi_2 \qquad \Psi_{IN1}(\mathtt{f}) = \Psi_{IN2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN1}) \cap \mathsf{dom}(\Psi_{IN2})\end{array}}{\Psi_{IN1} \cup \Psi_{IN2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2}\ (\textsc{link})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}$ **(*Well-formed Instruction Sequence*)**

$$\frac{\mathtt{a} \Rightarrow_{\mathsf{c}} \mathtt{a}' \qquad \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{mov}, \mathsf{movi}, \mathsf{alloc}, \mathsf{free}, \mathsf{ld}, \mathsf{st}\}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{c}; \mathbb{I}}\ (\textsc{seq})$$

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jd}\ \mathtt{f}}\ (\textsc{jd})$$

$$\frac{\begin{array}{ll}(\lambda(\mathbb{H},\mathbb{R}).\, \mathbb{R}(\mathtt{r}_s) \le i \ \wedge\ \mathtt{a}\ (\mathbb{H},\mathbb{R})) \Rightarrow \mathtt{a}' & \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I} \\ (\lambda(\mathbb{H},\mathbb{R}).\, \mathbb{R}(\mathtt{r}_s) > i \ \wedge\ \mathtt{a}\ (\mathbb{H},\mathbb{R})) \Rightarrow \Psi(\mathtt{f}) & \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}}\ (\textsc{bgti})$$

$$\frac{\mathtt{a} \ \Rightarrow\ (\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}'(\mathbb{H},\mathbb{R}) \ \wedge\ \mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\Psi) \ \wedge\ \Psi(\mathbb{R}(\mathtt{r})) = \mathtt{a}')}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp})$$

Figure 2.5: Inference rules of CAP

the inference rules for well-formed code heaps from the original CAP. A module is defined as a small code heap which can contain as few as one code block. Each module is associated with an "import" Ψ_{IN} interface and an "export" interface Ψ. A programmer first establishes well-formedness of each individual module via the CDHP rule. Two non-conflicting modules can then be linked together via the LINK rule. All code blocks will eventually be linked together to form a single global code heap with specification Ψ_G (which is used in the well-formed program rule). These two code heap rules provide basic support for modular verification. However, as we will show in the next Chapter, modularity breaks down when we reason about first-class code pointers passing across the module's boundary.

The intuition behind well-formed instruction sequence judgment is that if the instruction sequence $\mathbb{I}$ starts execution in a machine state which satisfies assertion $\mathtt{a}$, then executing $\mathbb{I}$ is safe with respect to the specification Ψ. An instruction sequence preceded by $\mathtt{c}$ is safe (rule SEQ) if we can find another assertion $\mathtt{a}'$ which serves both as the postcondition of $\mathtt{c}$, (that is, $\mathtt{a}'$ holds on the updated machine state after executing $\mathtt{c}$), and as the precondition of the tail instruction sequence.

A direct jump is safe (rule JD) if the current assertion can imply the precondition of the target code block as specified in Ψ. An indirect jump is similar (rule JMP) except that it refers to the register file for the target code label; unfortunately, this treatment of first class code pointers requires reasoning about global control flow and breaks the modularity (see the next chapter).

A programmer's task, when proving the well-formedness of a code block, involves mostly applying the appropriate inference rules, finding intermediate assertions like $\mathtt{a}'$, and proving all the assertion subsumption relations (which are implemented as logical implications in the mechanized meta logic).

Soundness. The soundness theorem below guarantees that given a well-formed CAP program, starting with its current instruction sequence, the machine will never get stuck.

Theorem 2.1 (CAP Soundness)

If $\Psi_{G} \vdash \{\texttt{a}\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^{n} \mathbb{P}'$.

The proof for this theorem can be established following the syntactic approach of proving type soundness [59] by proving the progress and preservation lemmas.

Lemma 2.2 (CAP Progress)

If $\Psi_{G} \vdash \{\texttt{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Lemma 2.3 (CAP Preservation)

If $\Psi_{G} \vdash \{\texttt{a}\}\,\mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$ then there exists an assertion $\texttt{a}'$ such that $\Psi_{G} \vdash \{\texttt{a}'\}\,\mathbb{P}'$.

In order to address the new LINK rule, we need the following code heap typing lemma, whose proof needs the instruction sequence weakening lemma below.

Lemma 2.4 (CAP Code Heap Typing)

If $\Psi_{IN} \vdash \mathbb{C} : \Psi$ and $\texttt{f} \in \mathsf{dom}(\Psi)$, then $\texttt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\Psi(\texttt{f})\}\,\mathbb{C}(\texttt{f})$.

Lemma 2.5 (CAP Instruction Sequence Weakening)

If $\Psi \vdash \{\texttt{a}\}\,\mathbb{I}$, $\Psi \subseteq \Psi'$, and $\texttt{a}' \Rightarrow \texttt{a}$ then $\Psi' \vdash \{\texttt{a}'\}\,\mathbb{I}$.

Yu *et al* [63, 64] have also shown that CAP can be easily extended to prove more general safety properties by introducing invariant assertions into the inference rules. Furthermore, by mechanizing the CAP inference rules and the soundness proofs in Coq, we can easily construct FPCC packages for CAP programs [63, 28].

There have been many works following the CAP framework: dynamic storage allocation [63], interfacing with TAL [28], concurrent verification [64, 20], support of embedded code pointers [47] (to be discussed in this dissertation), stack-based control abstraction [21], and open framework for interoperation [19].

Chapter 3

Embedded Code Pointers and the Basic XCAP Framework

In this chapter we present an important problem with CAP and other logic-based verification methods: embedded code pointers (ECP). Being a crucial concept for programming, the lack of modular and expressive ECP support prevents these methods from being used to certify realistic system kernel code. We then present our solution for the ECP problem, a new XCAP framework evolved from CAP. We first discuss the idea of "extended propositions," which can be roughly viewed as a mixture of logic formulas and syntactic constructs. We then discuss the basic structure of the new XCAP framework and show how to use syntactic techniques to perform modular reasoning on ECPs while still retaining the expressive power of Hoare logic. XCAP shares the same target machine TM (see Figure 2.1 and 2.2) with CAP.

XCAP provides the first simple and general solution to the ECP problem for Hoare-logic verification systems. Here, by "simple," we mean that our method does not alter the structure of Hoare-style program derivations and assertions. By "general", we mean that our technique can truly handle all kinds of machine-level ECPs, including those hidden inside higher-order closures (together with next chapter).

3.1 Embedded Code Pointers

ECPs, based on context and time, are often referred to as computed-gotos, stored procedures, higher-order functions, indirect jumps, continuation pointers, and so on. As the variations of its name suggest, ECP has long been an extensively used concept in programming. Because of the ECP problem, PCC systems based on Hoare logic have to either avoid supporting indirect jumps [44, 64], limit the assertions (for ECPs) to types only [28], sacrifice the modularity by requiring whole-program reasoning [63], or resort to construction of complex semantic models [9, 2]. In Reynolds [54], supporting ECPs is listed as one of the main open problems for separation logic.

At the assembly level (as in TM), ECPs denote those memory addresses (labels) stored in registers or memory cells, pointing to the start of code blocks. A special kind of ECPs is the function return addresses; more general kinds of ECPs can also be found in closures. Supporting ECPs is an essential part of assembly code verification. To understand better the ECP problem, let's take a look at the following example:

```
f1: mov r1, f1   // no assumption
    jd  f2

f2: jmp r1       // r1 stores an ECP with no assumption
```

Here we have defined two assembly code blocks. The first block, labeled $\mathtt{f}_1$, makes no assumption about the state; it simply moves the code label $\mathtt{f}_1$ into register $\mathtt{r}_1$ and then directly jumps to the other code block labeled $\mathtt{f}_2$. The $\mathtt{f}_2$ block requires that upon entering, register $\mathtt{r}_1$ must contain a code pointer that has no assumption about the state; it simply makes an indirect jump to this embedded code pointer and continues execution from there. If the execution initially starts from $\mathtt{f}_1$, the machine will loop forever between the two code blocks and never get stuck. It is important to note that both code blocks are independently written so they are expected to not only just work with each other, but with any other code satisfying their specifications as well.

We introduce a predicate, $\mathsf{cptr}(\mathtt{f}, \mathtt{a})$, to state that value $\mathtt{f}$ is a *valid* code pointer with precondition $\mathtt{a}$. Following the notations in Section 2.3, we define the "no assumption" assertion as

$$\mathsf{a}_T \triangleq \lambda\mathbb{S}.\,\mathsf{True}.$$

The preconditions for $\mathtt{f}_1$ and $\mathtt{f}_2$ can be written as

$$\mathsf{a}_1 \triangleq \mathsf{a}_T$$

and

$$\mathsf{a}_2 \triangleq \lambda(\mathbb{H},\mathbb{R}).\,\mathsf{a}_T\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}_1),\mathsf{a}_T).$$

But what should be the definition of $\mathsf{cptr}(\mathtt{f},\mathsf{a})$?

Semantic approach. The semantic approach to this problem is to directly internalize the Hoare derivations as part of the assertion language and to define $\mathsf{cptr}(\mathtt{f},\mathsf{a})$ as valid if there exists a Hoare derivation for the code block at $\mathtt{f}$ with precondition a. Using the notation from CAP, it can be informally written as:

$$\mathsf{cptr}(\mathtt{f},\mathsf{a})\ \triangleq\ \Psi\vdash\{\mathsf{a}\}\,\mathbb{C}(\mathtt{f}).$$

This is clearly ill-formed since Ψ is not defined anywhere and it can not be treated as a parameter of the cptr predicate—the assertion a, which is used to form Ψ, may refer to the cptr predicate again.

Semantic approach: stratification. To break the circularity, one approach [49, 42] is to internalize Hoare-logic derivations as part of the assertion language and stratify all ECPs (as well as the code heaps and specifications that contain them) so that only the cptr definitions (and well-formedness proofs) for highly-ranked code blocks can refer to those of lower-ranked ones. More specifically, a first order code pointer does not specify any ECP in its precondition (its code does not make any indirect jump) so we can define cptr over them first; an n-th order code pointer can only refer to those lower-order ECPs so we can define cptr inductively following the same order. The cptr predicate definition becomes:

$$\mathsf{cptr}(\mathtt{f},\mathsf{a},k) \triangleq \Psi_{k-1} \vdash \{\mathsf{a}\}\, \mathbb{C}_{k-1}(\mathtt{f}).$$

While the typing structures of well-formed instruction sequences would look like below.

$$\dfrac{\dfrac{\dfrac{\Psi_0 \vdash \{\mathsf{a}_0\}\ldots;\mathsf{jd}\ \ldots}{\vdots} \qquad \ldots}{\Psi_{n-1} \vdash \{\mathsf{a}_{n-1}\}\ldots;\mathsf{jmp}\ \ldots} \qquad \ldots}{\Psi_n \vdash \{\mathsf{a}_n\}\ldots;\mathsf{jmp}\ \ldots}$$

Stratification works for monomorphic languages with simple procedure parameters [49, 42], however, for machine-level programs where ECPs can appear in any part of the memory, tracking the orders of ECPs is impossible: ECPs can appear on the stack or in another function's closure (often hidden because closures are usually existentially quantified [39]).

Semantic approach: indexing. Another approach, by Appel *et al* [9, 2, 56], also introduces an parameter k to the cptr predicate. Instead of letting k refer to the depth of nesting ECPs as the stratification approach does, the "index" k now refers to the maximum number of safe future computation steps. Roughly speaking, $\mathsf{cptr}(\mathtt{f},\mathsf{a},k)$ means that it is "safe" to execute the next (at most) $k-1$ instructions, starting from the code block at $\mathtt{f}$ with precondition a.

$$\mathsf{cptr}(\mathtt{f},\mathsf{a},k) \triangleq \forall i<k.\, \Psi \vdash_i \{\mathsf{a}\}\mathbb{C}(\mathtt{l})$$

Note that here a and Ψ are **not** defined as state predicates and mapping from code labels to them. Instead, indexing must also be done for assertions and all the Hoare inference rules. For example, the indirect jump rule would have the following shape:

$$\frac{\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R}).\, \mathsf{a}'\ (\mathbb{H},\mathbb{R}) \wedge \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}',k-1)}{\Psi \vdash_k \{\mathsf{a}\}\mathsf{jmp}\ \mathtt{r}}\ .$$

Indexed assertions can only describe safety properties of finite steps. To establish safety properties about infinite future executions, one needs to do induction over the index. Because of the pervasive uses of indices everywhere, indexing dramatically alters the structure of Hoare-logic program derivations and assertions. This makes it hard to use together with other extensions such as separation logic [54] and rely-guarantee-based reasoning [64, 20].

Syntactic approach. Rather than using the semantic methods, CAP takes a syntactic approach and is essentially reasoning about the control flow. Validity of ECPs is established in two steps. In the first step, in indirect jump rule JMP it only requires that we look up the assertion for the target code label stored in register $\mathtt{r}$ from the code heap specification Ψ. (The equality of assertions used here is the Coq equality $\mathtt{eq}$ which is equivalent to the Leibniz' equality.)

$$\frac{\mathsf{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, \mathsf{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r})) = \mathsf{a}')}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ .$$

Then in the top-level PROG rule the well-formedness of global code heap is checked ($\Psi_G \vdash \mathbb{C} : \Psi_G$) to make sure that every assertion stored in Ψ_G (which is the union of all local Ψ) is indeed a valid precondition for the corresponding code block. The effect of these two steps, combined together, guarantees that the Hoare derivations internalized in the semantic approach is still obtainable in the syntactic approach for the preservation of the whole program. This approach is often used by type systems such as TAL.

But how do we know that such label indeed falls into the domain of Ψ? We reason about the control flow. Take the code block of $\mathtt{f}_2$ for example, we need to prove:

$$\mathsf{a}_2 \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, \mathsf{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathbb{R}(\mathtt{r}_1) \in \mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r}_1)) = \mathsf{a}')$$

which unfolds into

$$\forall \mathbb{H},\mathbb{R}.\, (\mathsf{a}_T\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}_1),\mathsf{a}_T)) \supset (\mathsf{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\ \mathbb{R}(\mathtt{r}_1) \in \mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r}_1)) = \mathsf{a}').$$

Clearly we should require the following to hold:

$$\forall \mathbb{H},\mathbb{R}.\, \mathsf{cptr}(\mathbb{R}(\mathtt{r}_1),\mathsf{a}_T) \supset (\mathbb{R}(\mathtt{r}_1) \in \mathsf{dom}(\Psi)\ \wedge\ \Psi(\mathbb{R}(\mathtt{r}_1)) = \mathsf{a}').$$

If we let the cptr predicate directly refer to Ψ in its definition, assertion and specification become cyclic definitions since Ψ consists of a mapping from code labels to assertions (which can contain cptr).

Previous CAP implementation [63] transforms the above formula into:

$$\forall \mathbb{H}, \mathbb{R}.\, \mathsf{cptr}(\mathbb{R}(\mathtt{r}_1), \mathtt{a}_T) \supset \mathbb{R}(\mathtt{r}_1) \in \{\, \mathtt{f} \mid \mathtt{f} \in \mathsf{dom}(\Psi) \wedge \Psi(\mathtt{f}) = \mathtt{a}_T \}$$

and statically calculates the address set on the right side using the global code heap specification Ψ_G. It can then define:

$$\mathsf{cptr}(\mathtt{f}, \mathtt{a}) \triangleq \mathtt{f} \in \{\, \mathtt{f}' \mid \mathtt{f}' \in \mathsf{dom}(\Psi_G) \wedge \Psi_G(\mathtt{f}') = \mathtt{a} \}.$$

This is clearly not satisfactory as the actual definition of $\mathsf{cptr}(\mathtt{f}, \mathtt{a})$ no longer refers to $\mathtt{a}$! Instead, it will be in the form of $\mathtt{f} \in \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$, which is not modular and very hard to reason about.

Taking the modularity issue more seriously, from the JMP rule again, we notice that all ECPs it can jump to are those contained in the current local Ψ only. Since the CAP language presented in previous section supports separate verification through linking rule LINK, when checking each module's code heap, we do not have the global specification Ψ_G and only have the declared import interface Ψ_{IN}. For our example, checking the code blocks under different organizations of modules would result in different $\mathtt{a}_2$ assertions:

$\lambda(\mathbb{H}, \mathbb{R}).\, \mathbb{R}(\mathtt{r}_1) \in \{\mathtt{f}_1\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are in a same module
and there is no other block with precondition $\mathtt{a}_T$ in it;

$\lambda(\mathbb{H}, \mathbb{R}).\, \mathbb{R}(\mathtt{r}_1) \in \{\mathtt{f}_1, \mathtt{f}_3\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are in a same module
and there is a block $\mathtt{f}_3$ also with precondition $\mathtt{a}_T$ in it;

$\lambda(\mathbb{H}, \mathbb{R}).\, \mathbb{R}(\mathtt{r}_1) \in \{\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are not in a same module
and there is no other block with precondition $\mathtt{a}_T$ in $\mathtt{f}_2$ module;

$\lambda(\mathbb{H}, \mathbb{R}).\, \mathbb{R}(\mathtt{r}_1) \in \{\mathtt{f}_3\}$ when $\mathtt{f}_1$ and $\mathtt{f}_2$ are not in a same module
and there is a block $\mathtt{f}_3$ also with precondition $\mathtt{a}_T$ in $\mathtt{f}_2$ module.

Since we usually do not know the target addresses statically, we cannot put all possible indirect code pointers into the import specification Ψ_{IN}. The syntactic approach used by CAP cannot support general ECPs without resorting to the whole-program analysis. This greatly limits CAP's modularity and expressive power.

Challenges. Given the ECP problem explained so far, we can summarize some important criteria for evaluating its possible solutions:

- Is it expressive? The new system should retain all expressive power from CAP. In particular, we still want to write assertions as general logic predicates. Ideally, there should be a "type-preserving" translation from CAP into the new system.
- Is it easy to specify? The specifications should be self-explanatory and can be used to reason about safety properties directly. There should be no need of translating ECP specifications into less informative forms such as indexed assertions or addresses sets as found in approaches aforementioned.
- Is it modular? The specifications should be independently writable and ECPs can be freely passed across the modular boundaries.
- Is it simple? The approach should better not involve overwhelming efforts in design and implementation when compared to CAP. It should not alter or pollute the basic structure of Hoare-style program derivations and assertions.
- Can it support extensions easily? The approach should work smoothly with common language features and popular extensions to Hoare logic.

3.2 Extended Propositions

To avoid the "circular specification" problem in the syntactic approach (to ECP), we break the loop by adding a small amount of syntax to the assertion language, and then split the syntax of assertions from their "meanings" (or validities). The key idea here is to delay the checking of the well-formedness property of ECPs. Instead of checking them individually at the instruction sequence level using locally available specification Ψ_{IN}, we collect all these checks into one global condition that would only need to be established with respect to the global code heap specification Ψ_G when all code are finally linked together before execution.

Basically cptr is now merely a syntactic constant and can appear in any assertion at any time in the form of $\mathsf{cptr}(\mathtt{f},\mathtt{a})$. Its meaning (or validity) is not revealed during the verification of local modules (*i.e.*, the well-formed instruction sequence rules). An "interpretation" will translate the ECP assertion syntax into its meaning (as a meta proposition) when the global code heap specification Ψ_G is finally available in the top-level PROG rule. The PROG rule will then complete the well-formedness check for the whole program. We achieve modularity through this two-stage verification structure.

Note that requiring the global specification Ψ_G in the PROG rule does not break any modularity, since at runtime all code (and their specifications) must be made available before they can be executed. Besides, the top-level rule PROG only needs to be validated once for the initial machine configuration.

Extended propositions. Figure 3.1 defines *extended logical propositions* (*PropX*). *PropX* can be viewed as a lifted version of the meta logic propositions, extended with an additional cptr constant: the base case $\langle p \rangle$ is just the lifted proposition p (thus *PropX* retains the full expressive power of meta logic propositions); cptr is the constructor for specifying ECP propositions. To interoperate lifted propositions and ECP propositions, we also lift all the logical connectives and quantifiers.

For the universal and existential quantifications, we use higher-order abstract syntax (HOAS) [52] to represent them. For example, $\forall\!\!\forall x{:}A.\mathtt{P}$ is actually implemented as $\forall\!\!\forall(\lambda x{:}A.\mathtt{P})$. The benefit here is that we can utilize the full expressive power of the mechanized meta logic (CiC/Coq) and use a single quantifier to quantify over all possible types (A) such as *Prop*, *State*, and even *State* → *Prop*.

$$
\begin{array}{rrcll}
(\mathit{PropX}) & \mathtt{P},\mathtt{Q} & ::= & \langle p \rangle & \mathit{lifted\ meta\ proposition} \\
 & & | & \mathsf{cptr}(\mathtt{f},\mathtt{a}) & \mathit{embedded\ code\ pointer} \\
 & & | & \mathtt{P} \mathbin{\wedge\!\!\wedge} \mathtt{Q} & \mathit{conjunction} \\
 & & | & \mathtt{P} \mathbin{\vee\!\!\vee} \mathtt{Q} & \mathit{disjunction} \\
 & & | & \mathtt{P} \twoheadrightarrow \mathtt{Q} & \mathit{implication} \\
 & & | & \forall\!\!\!\forall x\!:\!A.\,\mathtt{P} & \mathit{universal\ quantification} \\
 & & | & \exists\!\!\!\exists x\!:\!A.\,\mathtt{P} & \mathit{existential\ quantification} \\
(\mathit{Assertion}) & \mathtt{a} & \in & \mathit{State} \rightarrow \mathit{PropX} &
\end{array}
$$

Figure 3.1: Extended propositions

$$
\begin{array}{rcl}
[\![\, \langle p \rangle \,]\!]_{\Psi} & \triangleq & p \\
[\![\mathsf{cptr}(\mathtt{f},\mathtt{a})]\!]_{\Psi} & \triangleq & \mathtt{f} \in \mathsf{dom}(\Psi) \wedge \Psi(\mathtt{f}) = \mathtt{a} \\
[\![\mathtt{P} \mathbin{\wedge\!\!\wedge} \mathtt{Q}]\!]_{\Psi} & \triangleq & [\![\mathtt{P}]\!]_{\Psi} \wedge [\![\mathtt{Q}]\!]_{\Psi} \\
[\![\mathtt{P} \mathbin{\vee\!\!\vee} \mathtt{Q}]\!]_{\Psi} & \triangleq & [\![\mathtt{P}]\!]_{\Psi} \vee [\![\mathtt{Q}]\!]_{\Psi} \\
[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_{\Psi} & \triangleq & [\![\mathtt{P}]\!]_{\Psi} \supset [\![\mathtt{Q}]\!]_{\Psi} \\
[\![\forall\!\!\!\forall x\!:\!A.\,\mathtt{P}]\!]_{\Psi} & \triangleq & \forall B\!:\!A.\, [\![\mathtt{P}[B/x]]\!]_{\Psi} \\
[\![\exists\!\!\!\exists x\!:\!A.\,\mathtt{P}]\!]_{\Psi} & \triangleq & \exists B\!:\!A.\, [\![\mathtt{P}[B/x]]\!]_{\Psi}
\end{array}
$$

Figure 3.2: Interpretations of extended propositions

Extended propositions can be used to construct "extended" predicates using the abstraction facility in the underlying meta logic. For example, the following extended state predicate of type *State*→*PropX* says registers $\mathtt{r}_1$ and $\mathtt{r}_2$ store the same value, while $\mathtt{r}_3$ stores a non-NULL value:

$$\lambda(\mathbb{H},\mathbb{R}).\, \langle \mathbb{R}(\mathtt{r}_1) = \mathbb{R}(\mathtt{r}_2) \wedge \mathbb{R}(\mathtt{r}_2) \neq \mathsf{NULL} \rangle.$$

Extended predicates are not limited to be over machine states only. For example, the following extended value predicate resembles the code pointer type found in type systems. (Here, $\mathtt{a}$, the precondition of the code pointed to by $\mathtt{f}$, is an extended state predicate.)

$$\mathsf{code}\ \mathtt{a} \triangleq \lambda \mathtt{f}.\, \mathsf{cptr}(\mathtt{f},\mathtt{a})$$

Extended propositions adds a thin layer of syntax over meta propositions. Figure 3.2

presents an **interpretation** of their validity in meta logic. It is defined as a meta function. A lifted proposition $\langle p \rangle$ is valid if p is valid in the meta logic. Validity of ECP propositions can only be testified with a code heap specification Ψ (formally defined in the next section), so we make it a parameter of the interpretation function; Ψ is typically instantiated by (but not limited to) the global specification Ψ_G. Interpretation of $\mathsf{cptr}(\mathtt{f}, \mathtt{a})$ tests the equality of $\mathtt{a}$ with $\Psi(\mathtt{f})$. Here we use the inductively defined Coq equality `eq` (equivalent to the Leibniz' equality) extended with extensionality of functions (a safe extension commonly used in the Coq [30] community). Note that the use of equality predicate in logic is different from the use of equality function in programming—a programmer must supply the proper equality proofs in order to satisfy the ECP interpretation. Extended logical connectives and quantifiers are interpreted in a straight-forward way. Note that HOAS [52] is used in the interpretation of extended implications.

Interpretation of assertions can be trivially defined as

$$[\![\mathtt{a}]\!]_\Psi \ \triangleq\ \lambda \mathbb{S}.\, [\![\mathtt{a}\, \mathbb{S}]\!]_\Psi$$

which turns "syntactic" program specifications in extended propositions into "real" program invariants in the mechanized meta logic.

3.3 The XCAP Framework

In this section, we present a new XCAP framework, which is based on CAP and uses *PropX* as its assertion language. We refer readers to Section 2.3 for more details on the CAP framework and only highlight the difference between XCAP and CAP in this section.

Assertion Language. We present the assertion language of XCAP in Figure 3.3. The only difference between it and Figure 2.4 is that assertions ($\mathtt{a}$) are now defined as extended state predicates. We also define code heap specifications (Ψ) and the assertion subsumption relation ($\Rightarrow$) accordingly. Note that assertions are first turned into meta logical interpretations before doing subsumptions.

$$\begin{array}{rccl}
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \leadsto \mathtt{a}\}^{*} \\
(\mathit{Assertion}) & \mathtt{a} & \in & \mathit{State} \rightarrow \mathit{PropX} \\
(\mathit{AssertImp}) & \mathtt{a} \Rightarrow \mathtt{a}' & \triangleq & \forall \Psi, \mathbb{S}.\, \llbracket \mathtt{a} \rrbracket_{\Psi}\, \mathbb{S} \supset \llbracket \mathtt{a}' \rrbracket_{\Psi}\, \mathbb{S} \\
(\mathit{StepImp}) & \mathtt{a} \Rightarrow_{\mathtt{C}} \mathtt{a}' & \triangleq & \forall \Psi, \mathbb{S}.\, \llbracket \mathtt{a} \rrbracket_{\Psi}\, \mathbb{S} \supset \llbracket \mathtt{a}' \rrbracket_{\Psi}\, \mathtt{Next}_{\mathtt{C}}(\mathbb{S})
\end{array}$$

Figure 3.3: Assertion language of XCAP

Inference rules. To reason about TM programs in XCAP, just as we did for CAP in Figure 2.5, we present a similar set of inference rules for well-formed programs, code heaps, and instruction sequences in Figure 3.4. Other than the differences in the assertion language, we only modified the PROG rule and the JMP rule, and added a new ECP rule for introducing new ECP propositions into assertions on the fly. For the unchanged rules, readers can refer to Section 2.3 for explanations.

The change for the PROG rule is minor but important. Here we use the global specification Ψ_G in the interpretation of assertions which may contain ECP propositions. For state $\mathbb{S}$ to satisfy assertion $\mathtt{a}$, we require a proof for the meta proposition $(\llbracket \mathtt{a} \rrbracket_{\Psi_G}\, \mathbb{S})$. This is the only place in the XCAP inference rules where validity of assertions (with ECP propositions) needs to be established. Other rules only require subsumption between assertions.

For the JMP rule, instead of looking up the target code blocks' preconditions $\mathtt{a}'$ from the current (local) specification Ψ, we require the current precondition $\mathtt{a}$ to guarantee that the target code label $\mathbb{R}(\mathtt{r})$ is a valid ECP with $\mathtt{a}'$ as its precondition. Combined with the $(\llbracket \mathtt{a} \rrbracket_{\Psi_G}\, \mathbb{S})$ condition established in the PROG rule, we can deduce that $\mathtt{a}'$ is indeed the one specified in Ψ_G.

If we call the JMP rule "consumer" of ECP propositions, then the ECP rule can be called "producer." It is essentially a "cast" rule—it allows us to introduce new ECP propositions $\mathsf{cptr}(\mathtt{f}, \Psi(\mathtt{f}))$ about any code label $\mathtt{f}$ found in the current code heap specification Ψ into the new assertion $\mathtt{a}'$. This rule is often used when we move a constant code label into a register to create an ECP.

$\boxed{\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}}$ **(*Well-formed Program*)**

$$\frac{\Psi_G \vdash \mathbb{C} : \Psi_G \qquad ([\![\mathtt{a}]\!]_{\Psi_G}\ \mathbb{S}) \qquad \Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi_G \vdash \{\mathtt{a}\}\,(\mathbb{C},\mathbb{S},\mathbb{I})}\ (\textsc{prog})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C} : \Psi}$ **(*Well-formed Code Heap*)**

$$\frac{\Psi_{IN} \vdash \{\mathtt{a}_i\}\,\mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \leadsto \mathbb{I}_1, \ldots, \mathtt{f}_n \leadsto \mathbb{I}_n\} : \{\mathtt{f}_1 \leadsto \mathtt{a}_1, \ldots, \mathtt{f}_n \leadsto \mathtt{a}_n\}}\ (\textsc{cdhp})$$

$$\frac{\begin{array}{c}\Psi_{IN\,1} \vdash \mathbb{C}_1 : \Psi_1 \qquad \Psi_{IN\,2} \vdash \mathbb{C}_2 : \Psi_2 \qquad \Psi_{IN\,1}(\mathtt{f}) = \Psi_{IN\,2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN\,1}) \cap \mathsf{dom}(\Psi_{IN\,2})\end{array}}{\Psi_{IN\,1} \cup \Psi_{IN\,2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2}\ (\textsc{link})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}$ **(*Well-formed Instruction Sequence*)**

$$\frac{\mathsf{a} \Rightarrow_{\mathsf{c}} \mathsf{a}' \qquad \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{mov}, \mathsf{movi}, \mathsf{alloc}, \mathsf{free}, \mathsf{ld}, \mathsf{st}\}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{c};\mathbb{I}}\ (\textsc{seq})$$

$$\frac{\mathsf{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jd}\ \mathtt{f}}\ (\textsc{jd})$$

$$\frac{\begin{array}{l}(\lambda(\mathbb{H},\mathbb{R}).\,\langle \mathbb{R}(\mathtt{r}_s) \leq i\rangle \ \wedge\!\!\wedge\ \mathsf{a}\ (\mathbb{H},\mathbb{R})) \Rightarrow \mathsf{a}' \qquad \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I} \\ (\lambda(\mathbb{H},\mathbb{R}).\,\langle \mathbb{R}(\mathtt{r}_s) > i\rangle \ \wedge\!\!\wedge\ \mathsf{a}\ (\mathbb{H},\mathbb{R})) \Rightarrow \Psi(\mathtt{f}) \quad \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f};\mathbb{I}}\ (\textsc{bgti})$$

$$\frac{\mathsf{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\,\mathsf{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\!\!\wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathsf{a}'))}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp})$$

$$\frac{(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f}, \Psi(\mathtt{f})) \wedge\!\!\wedge\, \mathsf{a}\ \mathbb{S}) \Rightarrow \mathsf{a}' \quad \mathtt{f} \in \mathsf{dom}(\Psi) \quad \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I}}{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}\ (\textsc{ecp})$$

Figure 3.4: Inference rules of XCAP

Example. Combining the JMP and ECP rules, ECP knowledge can be built up by one module at the time of ECP creation and be passed around and get used for indirect jumps in other modules. For example, given the following code where we assume register $\mathtt{r}_{30}$ contains the return value and register $\mathtt{r}_{31}$ contains the return code pointer:

```
plus: add r30, r0, r1;   // fun plus (a, b) = a + b
      jmp r31

app2: mov r3, r0;        // fun app2(f, a, b) = f(a, b)
      mov r0, r1;
      mov r1, r2;
      jmp r3
```

we can assign them with the following XCAP specifications and make safe (higher-order) function calls such as *app2(plus,1,2)*.

$$
\begin{aligned}
\{\textsf{plus} \leadsto \lambda(\mathbb{H},\mathbb{R}).\,\exists a,b,ret.\\
&\langle \mathbb{R}(\mathtt{r}_0)=a \wedge \mathbb{R}(\mathtt{r}_1)=b \wedge \mathbb{R}(\mathtt{r}_{31})=ret\rangle\\
&\Wedge \textsf{cptr}(ret,\ \ \lambda(\mathbb{H}',\mathbb{R}').\,\langle \mathbb{R}'(\mathtt{r}_{30})=a+b\rangle)\}
\end{aligned}
$$

$$
\begin{aligned}
\{\textsf{app2} \leadsto \lambda(\mathbb{H},\mathbb{R}).\,\exists f,a,b,ret.\\
&\langle \mathbb{R}(\mathtt{r}_1)=a \wedge \mathbb{R}(\mathtt{r}_2)=b \wedge \mathbb{R}(\mathtt{r}_0)=f \wedge \mathbb{R}(\mathtt{r}_{31})=ret\rangle\\
&\Wedge \textsf{cptr}(f,\ \ \lambda(\mathbb{H}',\mathbb{R}').\,\exists a',b',ret'.\\
&\qquad\qquad \langle \mathbb{R}'(\mathtt{r}_0)=a' \wedge \mathbb{R}'(\mathtt{r}_1)=b' \wedge \mathbb{R}'(\mathtt{r}_{31})=ret'\rangle\\
&\qquad\quad \Wedge \textsf{cptr}(ret',\ \ \lambda(\mathbb{H}'',\mathbb{R}'').\,\langle \mathbb{R}''(\mathtt{r}_{30})=a'+b'\rangle))\\
&\Wedge \textsf{cptr}(ret,\ \ \lambda(\mathbb{H}',\mathbb{R}').\,\langle \mathbb{R}'(\mathtt{r}_{30})=a+b\rangle)\}
\end{aligned}
$$

Soundness. The soundness of XCAP is proved in the same way as we did for CAP. We give the main lemmas and a proof sketch here. More details can be found in Appendix B.

Lemma 3.1 (XCAP Progress)

If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Proof Sketch: Suppose $\mathbb{P}=(\mathbb{C},\mathbb{S},\mathbb{I})$, by inversion we obtain $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}$. The proof is by induction over this derivation. ■

Lemma 3.2 (XCAP Preservation)

If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$ then there exists an assertion $\mathtt{a}'$ such that $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{P}'$.

Proof Sketch: Suppose $\mathbb{P} = (\mathbb{C}, \mathbb{S}, \mathbb{I})$; by inversion we obtain $\Psi_G \vdash \mathbb{C} : \Psi_G$, $([\![\mathtt{a}]\!]_{\Psi_G}\ \mathbb{S})$, and $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}$. We do induction over derivation $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}$. The only interesting cases are the JMP and ECP rules.

For the JMP rule case, let $\mathbb{S}$ be $(\mathbb{H}, \mathbb{R})$. By the implication $\mathsf{a} \Rightarrow (\lambda(\mathbb{H}, \mathbb{R}).\, \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}'))$ and the interpretation of cptr it follows that $\mathtt{a}' = \Psi_G(\mathbb{R}(\mathtt{r}))$ and $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\Psi_G)$. Then by the same code heap typing lemma as discussed in Section 2.3, it follows that $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{C}(\mathbb{R}(\mathtt{r}))$. Finally by $\mathsf{a} \Rightarrow \mathtt{a}'$ it follows that $\mathtt{a}'(\mathbb{H}, \mathbb{R})$.

For the ECP case, by the code heap typing lemma and by $(\lambda\mathbb{S}.\, \mathsf{cptr}(\mathtt{f}, \Psi_G(\mathtt{f})) \mathbin{\mathbb{\wedge}} \mathsf{a}\ \mathbb{S}) \Rightarrow \mathtt{a}'$ it follows that $[\![\mathtt{a}'\ \mathbb{S}]\!]_{\Psi_G}$. Also we have $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}$. Then we use the induction hypodissertation to finish the proof. ■

Theorem 3.3 (XCAP Soundness)

If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

3.4 Discussion

The main idea of XCAP is to support Hoare-style reasoning of ECPs by extending the assertion language with a thin layer of syntax. Next we review XCAP using the criteria given in Section 3.1. The following theorem presents a simple "type-preserving" translation from CAP to XCAP and shows that XCAP is at least as powerful as CAP. To avoid confusion, we use $\vdash_{CAP}$ and $\vdash_{XCAP}$ to represent CAP judgments (as defined in Figure 2.4) and XCAP ones (as defined in Figure 3.4). See our implementation for detailed proofs.

Theorem 3.4 (CAP to XCAP Translation)

We define the lifting of CAP assertions and specifications as:

$$\ulcorner \mathtt{a} \urcorner \triangleq \lambda\mathbb{S}.\, \langle \mathtt{a}\ \mathbb{S} \rangle$$

$$\text{and } \ulcorner \{\mathtt{f}_1 \leadsto \mathtt{a}_1, \ldots, \mathtt{f}_n \leadsto \mathtt{a}_n\} \urcorner \triangleq \{\mathtt{f}_1 \leadsto \ulcorner \mathtt{a}_1 \urcorner, \ldots, \mathtt{f}_n \leadsto \ulcorner \mathtt{a}_n \urcorner\}.$$

1. If $\Psi_G \vdash_{CAP} \{\mathtt{a}\}\,\mathbb{P}$ then $\ulcorner\Psi_G\urcorner \vdash_{XCAP} \{\ulcorner\mathtt{a}\urcorner\}\,\mathbb{P}$;

2. if $\Psi_{IN} \vdash_{CAP} \mathbb{C} : \Psi$ then $\ulcorner\Psi_{IN}\urcorner \vdash_{XCAP} \{\mathbb{C}\}\ulcorner\Psi\urcorner$;

3. if $\Psi \vdash_{CAP} \{\mathtt{a}\}\,\mathbb{I}$ then $\ulcorner\Psi\urcorner \vdash_{XCAP} \{\ulcorner\mathtt{a}\urcorner\}\,\mathbb{I}$.

Proof Sketch: (1) and (2) are straight forward and based on (3). For (3), as CAP's JMP rule only reasons about preconditions in the current Ψ, we use ECP rule on all pairs of code label and precondition in Ψ and use XCAP's JMP rule to finish the proof. ■

The specification written in XCAP assertion language is close to a typical Hoare assertion and thus is easy to write and reason about. From a user perspective, there is no need to worry about the "meaning" of the ECP propositions because they are treated abstractly almost all the time.

XCAP is still lightweight because the lifted propositions $\langle p \rangle$ and their reasoning are shallowly embedded into the meta logic, which is the same as CAP. The added component of ECP propositions as well as other lifted connectives and quantifiers are simple syntactic constructs and do not involve complex constructions.

As we will show in later chapters, the XCAP framework can be easily extended to support other language features and popular extensions of Hoare logic.

Chapter 4

Impredicative Polymorphisms and Recursive Specifications

The XCAP system presented in the previous chapter enjoys great expressive power from its underlying meta logic. As the mini examples shown before, features such as data polymorphism can be easily supported. However, to support modular verification, when composing specifications, it is important to be able to abstract out and quantify over (part of) the specifications themselves. This is especially important for ECPs, since very often the specification for the target code is only partially disclosed to the callers. We extend the XCAP from the previous chapter to support this kind of impredicative polymorphism.

Using the extended XCAP, we solve the ECP problem for separation logic [54] and present a verification of a destructive list-append function listed as an example in [54].

Recursive specifications are very useful in describing complex invariants. Simple recursive data structures such as link-list are already supported by XCAP. However, for the recursive types ($\mu\alpha.\,\tau$) found in type systems, their counterpart in logic, "recursive predicates", can not be easily defined in XCAP. We follow the extension for impredicative polymorphisms and extended XCAP to support recursive specifications.

Both the extensions cause very little change in the XCAP inference rules and meta theory. Thus they are light-weight and proved sound.

4.1 Impredicative Polymorphisms and Recursive Specifications

Impredicative polymorphisms and recursive types can be easily supported in type systems. Take TAL [41] for example, it allows quantifications over value types, which correspond to value predicates in XCAP. Since XCAP predicates are much more flexible than types, we choose to support universal and existential quantifications over arbitrary extended predicates of type $A \rightarrow PropX$ (where A does not contain $PropX$). We reuse the quantifiers defined in XCAP in Section 3.3 and write **impredicative extended propositions** as follows:

$$\forall\!\!\forall \alpha : A \rightarrow PropX.\, \mathtt{P} \qquad \text{and} \qquad \exists\!\!\exists \alpha : A \rightarrow PropX.\, \mathtt{P}.$$

In the implementation of these impredicative quantifiers, the HOAS technique used for predicative ones no longer works because of the negative-occurrence restriction for inductive definitions. We use the de Bruijn notations [15] to encode the impredicative quantifiers. For more details, see Appendix 8.

The next task is to find a way to establish the **validity of impredicative extended propositions**. One obvious idea is to take the previously defined interpretation function in Figure 3.2 and directly apply it to the impredicative quantification cases as follows:

$$[\![\forall\!\!\forall \alpha : A \rightarrow PropX.\, \mathtt{P}]\!]_\Psi \triangleq \forall \mathtt{a} : A \rightarrow PropX.\, [\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$$
$$[\![\exists\!\!\exists \alpha : A \rightarrow PropX.\, \mathtt{P}]\!]_\Psi \triangleq \exists \mathtt{a} : A \rightarrow PropX.\, [\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$$

Unfortunately (but not surprisingly), the recursive call parameter $\mathtt{P}[\mathtt{a}/\alpha]$ may be larger than the original ones as $\mathtt{a}$ can bring in unbounded new sub-formulas. The interpretation function no longer terminates, and thus is not definable in the meta logic.

Our solution is to define the interpretation of extended propositions as the set of inductively defined validity rules shown in Figure 4.1. We define an environment of extended propositions as

$$(env) \quad \Gamma := \cdot \mid \Gamma, \mathtt{P}$$

$\boxed{\Gamma \vdash_{\Psi} \mathtt{P}}$ ***(Validity of Extended Propositions)***

(The following presentation omits the Ψ in judgment $\Gamma \vdash_{\Psi} P$.)

$$\frac{\mathtt{P} \in \Gamma}{\Gamma \vdash \mathtt{P}}\;(\textsc{env}) \qquad \frac{p}{\Gamma \vdash \langle p \rangle}\;(\langle\rangle\text{-}\textsc{i}) \qquad \frac{\Gamma \vdash \langle p \rangle \qquad p \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\;(\langle\rangle\text{-}\textsc{e})$$

$$\frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \vdash \mathsf{cptr}(\mathtt{f}, \mathtt{a})}\;(\textsc{cp-i}) \qquad \frac{\Gamma \vdash \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \qquad (\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\;(\textsc{cp-e})$$

$$\frac{\Gamma \vdash \mathtt{P} \qquad \Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \wedge\!\!\!\wedge \mathtt{Q}}\;(\wedge\!\!\!\wedge\text{-}\textsc{i}) \qquad \frac{\Gamma \vdash \mathtt{P} \wedge\!\!\!\wedge \mathtt{Q}}{\Gamma \vdash \mathtt{P}}\;(\wedge\!\!\!\wedge\text{-}\textsc{e}1) \qquad \frac{\Gamma \vdash \mathtt{P} \wedge\!\!\!\wedge \mathtt{Q}}{\Gamma \vdash \mathtt{Q}}\;(\wedge\!\!\!\wedge\text{-}\textsc{e}2)$$

$$\frac{\Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{P} \vee\!\!\!\vee \mathtt{Q}}\;(\vee\!\!\!\vee\text{-}\textsc{i}1) \qquad \frac{\Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \vee\!\!\!\vee \mathtt{Q}}\;(\vee\!\!\!\vee\text{-}\textsc{i}2) \qquad \frac{\Gamma \vdash \mathtt{P} \vee\!\!\!\vee \mathtt{Q} \qquad \Gamma, \mathtt{P} \vdash \mathtt{R} \qquad \Gamma, \mathtt{Q} \vdash \mathtt{R}}{\Gamma \vdash \mathtt{R}}\;(\vee\!\!\!\vee\text{-}\textsc{e})$$

$$\frac{\Gamma, \mathtt{P} \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q}}\;(\twoheadrightarrow\text{-}\textsc{i}) \qquad \frac{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \qquad \Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{Q}}\;(\twoheadrightarrow\text{-}\textsc{e})$$

$$\frac{\Gamma \vdash \mathtt{P}[B/x] \qquad \forall\, B : A}{\Gamma \vdash \forall\!\!\!\forall x : A.\,\mathtt{P}}\;(\forall\!\!\!\forall\text{-}\textsc{i}1) \qquad \frac{\Gamma \vdash \forall\!\!\!\forall x : A.\,\mathtt{P} \qquad B : A}{\Gamma \vdash \mathtt{P}[B/x]}\;(\forall\!\!\!\forall\text{-}\textsc{e}1)$$

$$\frac{B : A \qquad \Gamma \vdash \mathtt{P}[B/x]}{\Gamma \vdash \exists\!\!\!\exists x : A.\,\mathtt{P}}\;(\exists\!\!\!\exists\text{-}\textsc{i}1) \qquad \frac{\Gamma \vdash \exists\!\!\!\exists x : A.\,\mathtt{P} \qquad \Gamma, \mathtt{P}[B/x] \vdash \mathtt{Q} \qquad \forall\, B : A}{\Gamma \vdash \mathtt{Q}}\;(\exists\!\!\!\exists\text{-}\textsc{e}1)$$

$$\frac{\Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \qquad \forall\, \mathtt{a} : A \rightarrow PropX}{\Gamma \vdash \forall\!\!\!\forall \alpha : A \rightarrow PropX.\,\mathtt{P}}\;(\forall\!\!\!\forall\text{-}\textsc{i}2) \qquad \frac{\mathtt{a} : A \rightarrow PropX \qquad \Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha]}{\Gamma \vdash \exists\!\!\!\exists \alpha : A \rightarrow PropX.\,\mathtt{P}}\;(\exists\!\!\!\exists\text{-}\textsc{i}2)$$

Figure 4.1: Validity rules for impredicative extended propositions

The judgment, $\Gamma \vdash_{\Psi} \mathtt{P}$, means that $\mathtt{P}$ is valid under environment Γ and code heap specification Ψ. An extended proposition is valid if it is in the environment. Constructors of extended propositions have their introduction and elimination rules. The introduction rules of lifted proposition $\langle p \rangle$ and ECP proposition $\mathsf{cptr}(\mathtt{f}, \mathtt{a})$ require that p and $\Psi(\mathtt{f}) = \mathtt{a}$ be valid in the meta logic. Their elimination rules allow full meta-implication power in constructing derivations of validity of the new extended propositions. The rules for other constructors are standard and require little explanation.

The **interpretation of extended propositions** can be now be simply defined as their validity under the empty environment.

$$[\![\mathtt{P}]\!]_{\Psi} \triangleq \cdot \vdash_{\Psi} \mathtt{P}$$

Given the above definitions of interpretation and validity, we have proved the following soundness theorem (with respect to CiC/Coq) using the syntactic normalization proof method by Pfenning [51]. For proof details, see Appendix A.

Theorem 4.1 (Soundness of *PropX* Interpretation)

1. If $[\![\,\langle p\rangle\,]\!]_{\Psi}$ then p;
2. if $[\![\mathsf{cptr}(\mathtt{f},\mathtt{a})]\!]_{\Psi}$ then $\Psi(\mathtt{f}) = \mathtt{a}$;
3. if $[\![\mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}]\!]_{\Psi}$ then $[\![\mathtt{P}]\!]_{\Psi}$ and $[\![\mathtt{Q}]\!]_{\Psi}$;
4. if $[\![\mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q}]\!]_{\Psi}$ then either $[\![\mathtt{P}]\!]_{\Psi}$ or $[\![\mathtt{Q}]\!]_{\Psi}$;
5. if $[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_{\Psi}$ and $[\![\mathtt{P}]\!]_{\Psi}$ then $[\![\mathtt{Q}]\!]_{\Psi}$;
6. if $[\![\forall\!\!\!\forall x\!:\!A.\,\mathtt{P}]\!]_{\Psi}$ and $B\!:\!A$ then $[\![\mathtt{P}[B/x]\,]\!]_{\Psi}$;
7. if $[\![\,\exists\!\!\!\exists x\!:\!A.\,\mathtt{P}]\!]_{\Psi}$ then there exists $B\!:\!A$ such that $[\![\mathtt{P}[B/x]\,]\!]_{\Psi}$;
8. if $[\![\forall\!\!\!\forall \alpha\!:\!A \to \mathit{PropX}.\,\mathtt{P}]\!]_{\Psi}$ and $\mathtt{a}\!:\!A \to \mathit{PropX}$ then $[\![\mathtt{P}[\mathtt{a}/\alpha]\,]\!]_{\Psi}$;
9. if $[\![\,\exists\!\!\!\exists \alpha\!:\!A \to \mathit{PropX}.\,\mathtt{P}]\!]_{\Psi}$ then there exists $\mathtt{a}\!:\!A \to \mathit{PropX}$ such that $[\![\mathtt{P}[\mathtt{a}/\alpha]\,]\!]_{\Psi}$.

Corollary 4.2 (Consistency) $[\![\,\langle\mathsf{False}\rangle\,]\!]_{\Psi}$ is not provable.

To make impredicative extended propositions easy to use, in the implementation of *PropX* and its interpretation, we define additional concrete syntax and proof tactics to hide the de Bruijn representation and the interpretation detail. A user can mostly manipulate *PropX* objects in the same way as with *Prop* objects in Coq. See Chapter 8 for more details.

Inference rules and soundness. The XCAP indirect jump rule JMP from the one presented in Figure 3.4 can now be viewed as:

$$\frac{\mathsf{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, \exists \mathsf{a}'.\, (\mathsf{a}'\,(\mathbb{H},\mathbb{R}) \mathrel{\wedge\!\!\wedge} \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')))}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp}).$$

The existential quantification over the assertion a' (for the target code block) is moved from beging (implicitly) over the whole rule to being after the assertion subsumption ($\Rightarrow$). This change is important to support polymorphic code—the target assertion a' can now depend on the current assertion a.

All other inference rules of XCAP remain unchanged. The soundness of XCAP inference rules (Theorem 3.3) and the CAP to XCAP translation (Theorem 3.4) only need trivial modifications in the case of indirect jump. We do not restate them here.

Example. With impredicative quantifications, ECP can now be specified and used with great flexibility. For example, the *app*2 function in Section 3.3 can now be assigned with the following more general specification. Instead of being restricted to an argument with the "plus" functionality, any functions that take two arguments a and b and return a value satisfying (unrestricted) assertion $\mathsf{a}_{ret}(a,b)$ can be passed to *app*2.

$$\begin{aligned}
\{\mathsf{app2} \leadsto{} & \lambda(\mathbb{H},\mathbb{R}).\, \exists f,a,b,ret,\mathsf{a}_{ret}.\\
& \quad \langle \mathbb{R}(\mathtt{r}_1)=a \land \mathbb{R}(\mathtt{r}_2)=b \land \mathbb{R}(\mathtt{r}_0)=f \land \mathbb{R}(\mathtt{r}_{31})=ret \rangle\\
& \mathrel{\wedge\!\!\wedge} \mathsf{cptr}(f,\ \ \lambda(\mathbb{H}',\mathbb{R}').\, \exists a',b',ret'.\\
& \qquad\qquad \langle \mathbb{R}'(\mathtt{r}_0)=a' \land \mathbb{R}'(\mathtt{r}_1)=b' \land \mathbb{R}'(\mathtt{r}_{31})=ret' \rangle\\
& \qquad\quad \mathrel{\wedge\!\!\wedge} \mathsf{cptr}(ret',\ \ \mathsf{a}_{ret}(a',b')))\\
& \mathrel{\wedge\!\!\wedge} \mathsf{cptr}(ret,\ \ \mathsf{a}_{ret}(a,b))\}
\end{aligned}$$

Subtyping on ECP propositions. The ECP proposition has a very rigid interpretation. To establish the validity of $\mathsf{cptr}(\mathtt{f},\mathsf{a})$, $\Psi(\mathtt{f})$ must be "equal" to a. This is simple for the system, but is restrictive in usage and differs from typical type systems where subtyping can be used to relax code types. With the support of impredicative quantifications, instead of directly using cptr, we can define a more flexible predicate for ECPs:

$$\mathsf{codeptr}(\mathtt{f},\mathsf{a}) \triangleq \exists \mathsf{a}'.\, (\mathsf{cptr}(\mathtt{f},\mathsf{a}') \mathrel{\wedge\!\!\wedge} \forall\!\!\forall \mathbb{S}.\, \mathsf{a}\ \mathbb{S} \twoheadrightarrow \mathsf{a}'\ \mathbb{S}).$$

We can define the following subtyping lemma for ECP predicates.

Lemma 4.3 (Subtyping of ECP Propositions)

If $[\![\mathsf{codeptr}(\mathtt{f}, \mathsf{a}')]\!]_\Psi$ and $[\![\forall\!\!\!\forall \mathbb{S}.\, \mathsf{a}\,\mathbb{S} \twoheadrightarrow \mathsf{a}'\,\mathbb{S}]\!]_\Psi$ then $[\![\mathsf{codeptr}(\mathtt{f}, \mathsf{a})]\!]_\Psi$.

Proof: From $[\![\mathsf{codeptr}(\mathtt{f}, \mathsf{a}')]\!]_\Psi$ it follows that

$$[\![\exists\!\!\!\exists\, \mathsf{a}''.\, (\mathsf{cptr}(\mathtt{f}, \mathsf{a}'') \wedge\!\!\!\wedge \forall\!\!\!\forall \mathbb{S}.\, \mathsf{a}'\,\mathbb{S} \twoheadrightarrow \mathsf{a}''\,\mathbb{S})]\!]_\Psi.$$

By the soundness of interpretation theorem it follows that

$$\exists \mathsf{a}''.\, [\![\mathsf{cptr}(\mathtt{f}, \mathsf{a}'')]\!]_\Psi \wedge [\![\forall\!\!\!\forall \mathbb{S}.\, \mathsf{a}'\,\mathbb{S} \twoheadrightarrow \mathsf{a}''\,\mathbb{S}]\!]_\Psi.$$

Using the $\forall\!\!\!\forall$-I1, $\forall\!\!\!\forall$-E1, $\twoheadrightarrow$-I, and $\twoheadrightarrow$-E rules it follows that

$$[\![\forall\!\!\!\forall \mathbb{S}.\, \mathsf{a}\,\mathbb{S} \twoheadrightarrow \mathsf{a}''\,\mathbb{S}]\!]_\Psi.$$

Using the $\wedge\!\!\!\wedge$-I and $\exists\!\!\!\exists$-I2 rules it follows that

$$[\![\exists\!\!\!\exists\, \mathsf{a}''.\, (\mathsf{cptr}(\mathtt{f}, \mathsf{a}'') \wedge\!\!\!\wedge \forall\!\!\!\forall \mathbb{S}.\, \mathsf{a}\,\mathbb{S} \twoheadrightarrow \mathsf{a}''\,\mathbb{S})]\!]_\Psi.$$

Which is $[\![\mathsf{codeptr}(\mathtt{f}, \mathsf{a})]\!]_\Psi$. ■

4.2 Solving Reynolds's ECP Problem

Separation logic [54] is a recent Hoare-logic framework designed for reasoning about shared mutable data structures. Reynolds [54] listed supporting ECPs as a major open problem for separation logic. In this section, we show how to solve this problem within the XCAP framework (with impredicative polymorphisms support).

XCAP directly supports separation logic specifications and reasoning by defining their constructs and inference rules in the assertion language and meta logic as macros and lemmas. For example, the following are some separation logic primitives defined over the data heap (assuming $\uplus$ is the disjoint union):

$$
\begin{aligned}
\mathsf{emp} &\triangleq \lambda\mathbb{H}.\,\langle \mathsf{dom}(\mathbb{H}) = \{\}\rangle \\
\mathtt{l} \mapsto \mathtt{w} &\triangleq \lambda\mathbb{H}.\,\langle \mathsf{dom}(\mathbb{H}) = \{\mathtt{l}\} \land \mathbb{H}(\mathtt{l}) = \mathtt{w}\rangle \\
\mathtt{l} \mapsto \_ &\triangleq \lambda\mathbb{H}.\,\langle \exists \mathtt{w}.\,(\mathtt{l} \mapsto \mathtt{w}\ \mathbb{H})\rangle \\
\mathtt{a}_1 * \mathtt{a}_2 &\triangleq \lambda\mathbb{H}.\,\exists\,\mathbb{H}_1, \mathbb{H}_2.\,\langle \mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}\rangle \mathrel{\wedge\!\!\!\wedge} \mathtt{a}_1\ \mathbb{H}_1 \mathrel{\wedge\!\!\!\wedge} \mathtt{a}_2\ \mathbb{H}_2 \\
\mathtt{l} \mapsto \mathtt{w}_1, \ldots, \mathtt{w}_n &\triangleq \mathtt{l} \mapsto \mathtt{w}_1\ *\ \ldots\ *\ \mathtt{l}{+}n{-}1 \mapsto \mathtt{w}_n
\end{aligned}
$$

The frame rule can be defined as lemmas (derived rules) in XCAP:

$$
\frac{\mathtt{a}\ \Rightarrow\ (\mathtt{a}' \circ \mathtt{Next}_{\mathtt{c}})}{(\mathtt{a} * \mathtt{a}'')\ \Rightarrow\ ((\mathtt{a}' * \mathtt{a}'') \circ \mathtt{Next}_{\mathtt{c}})}\ (\textsc{frame-instr})
$$

$$
\frac{\{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\} \vdash \{\mathtt{a}\}\,\mathbb{I}}{\{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1 * \mathtt{a}', \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n * \mathtt{a}'\} \vdash \{\mathtt{a} * \mathtt{a}'\}\,\mathbb{I}}\ (\textsc{frame-iseq})
$$

ECP formulas can appear freely in these assertions and rules, thus it is very convenient to write specifications and reason about shared mutable data structures and embedded code pointers simultaneously. Note that it is assumed that in the derivations in the FRAME-ISEQ rule, there is no ECP or JMP rules being used.

Example: destructive list-append function in CPS. To demonstrate the above point, we verify a destructive version of the list-append example which Reynolds [54] used to define the ECP open problem. Following Reynolds, our destructive list-append function is written in continuation passing style (CPS):

```
append(x, y, rk) =
        if x == NULL then rk(y)
        else let k(z) = ([x+1] := z; rk(x))
             in append([x+1], y, k)
```

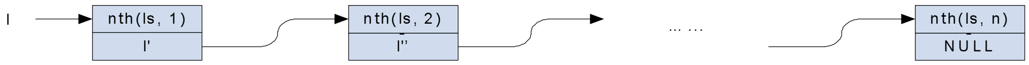
Here the *append* function takes three arguments: two lists *x* and *y* and a return continuation *rk*. If *x* is an empty list, it calls *rk* with list *y*. Otherwise, it first creates a new continuation function *k* which takes an (appended) list *z*, makes list *x*'s head node link to *z*, and passes the newly formed list (which is pointed to by *x*) to the return continuation *rk*. Variables *x* and *rk* form the closure environment for continuation function *k*. The *append* function then recursively calls itself with the tail of list x, list y, and the new continuation k. For node x, $[x]$ is its data and $[x+1]$ is the link to the next node.

We do closure conversion and translate *append* into TM assembly code. In the presentation, we often write $\mathtt{a}$ for assertion $\lambda(\mathbb{H},\mathbb{R}).\,\exists x_1{:}A_1,\ldots,x_n{:}A_n.\,\mathtt{a}$, so all free variables in $\mathtt{a}$ are existentially quantified right after the lambda abstraction. Formulas such as $(\mathtt{a} * \mathtt{a}'\ \mathbb{H})$ and $\mathbb{R}(\mathtt{r}) = \mathtt{w}$ are also simplified to be written as $\mathtt{a} * \mathtt{a}'$ and $\mathtt{r} = \mathtt{w}$.

Predicate $(\mathsf{list}\ ls\ \mathtt{l})$ describes a linked list pointed to by $\mathtt{l}$ where the data cell of each node stores the value in ls respectively. Here ls is a mathematical list where nil, $\mathtt{w} :: ls$ and $ls {+}{+} lt$ stand for the cases of empty list, cons, and append, respectively.

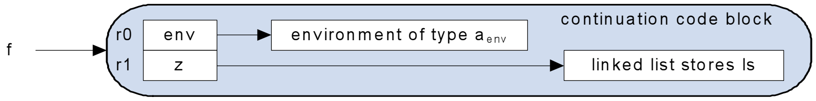
$$\mathsf{list}\ \mathsf{nil}\ \mathtt{l} \triangleq \mathsf{emp} \mathrel{\wedge\!\!\!\wedge} \langle \mathtt{l} = \mathsf{NULL} \rangle$$

$$\mathsf{list}\ (\mathtt{w} :: ls)\ \mathtt{l} \triangleq \exists \mathtt{l}'.\,\mathtt{l} \mapsto \mathtt{w}, \mathtt{l}' * \mathsf{list}\ ls\ \mathtt{l}'$$



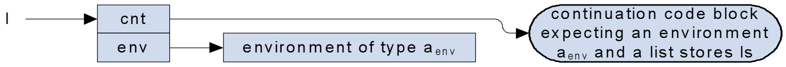
Predicate $(\mathsf{cont}\ \mathsf{a}_{env}\ ls\ \mathtt{f})$ requires $\mathtt{f}$ to point to a continuation code block which expects an environment of type a_{env} and a list which stores ls.

$$\mathsf{cont}\ \mathsf{a}_{env}\ ls\ \mathtt{f} \triangleq \mathsf{codeptr}(\mathtt{f},\ \lambda\mathbb{S}.\,\exists env, z.\ \langle \mathtt{r}_0 = env \wedge \mathtt{r}_1 = z \rangle \mathrel{\wedge\!\!\!\wedge} \mathsf{a}_{env}\ env * \mathsf{list}\ ls\ z)$$

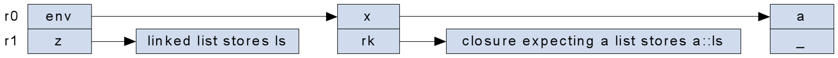


Predicate $(\mathsf{clos}\ ls\ \mathtt{l})$ describes a continuation closure pointed to by $\mathtt{l}$; this closure is a pair (cnt, env) where cnt is a continuation function pointer and env points to an environment for cnt. The environment predicate a_{env} is hidden inside the closure predicate.

$$\mathsf{clos}\ ls\ \mathtt{l} \triangleq \exists \mathsf{a}_{env}, cnt, env.\ \mathtt{l} \mapsto cnt, env * \mathsf{a}_{env}\ env \mathrel{\wedge\!\!\!\wedge} \mathsf{cont}\ \mathsf{a}_{env}\ ls\ cnt$$


l
cnt
env
environment of type a_env
continuation code block expecting an environment a_env and a list stores ls

In Figure 4.2 and Figure 4.3, we list the precondition for each instruction on its right side. The instruction determines which well-formed instruction rule to use at each step. State diagrams are drawn before all the interesting steps.



$\{\langle \mathtt{r}_0 = env \wedge \mathtt{r}_1 = z \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \mathsf{list}\ ls\ z \ast x \mapsto a, \_ \ast \mathsf{clos}\ (a :: ls)\ rk \ast env \mapsto x, rk\}$

```
k:      ld r2, r0(0)
```

$\{\langle \cdots\cdots\cdots \wedge \mathtt{r}_2 = x \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \cdots\cdots\cdots \ast env \mapsto \_, rk\}$

```
        ld r3, r0(1)
```

$\{\langle \cdots\cdots\cdots \wedge \mathtt{r}_3 = rk \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \cdots\cdots\cdots \ast env \mapsto \_, \_\}$

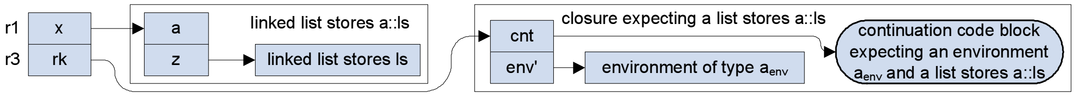
```
        free r0, 2
```

$\{\langle \cdots\cdots\cdots \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \cdots\cdots\cdots\}$

```
        st r2(1), r1
```

$\{\langle \cdots\cdots\cdots \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \cdots\cdots \ast x \mapsto a, z \ast \cdots\cdots\}$

```
        mov r1, r2
```


r1
r3
x
rk
a
z
linked list stores a::ls
linked list stores ls
closure expecting a list stores a::ls
cnt
env'
environment of type a_env
continuation code block expecting an environment a_env and a list stores a::ls

$\{\langle \mathtt{r}_1 = x \wedge \mathtt{r}_3 = rk \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \mathsf{list}\ (a :: ls)\ x \ast \mathsf{a}_{env}\ env' \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \mathsf{cont}\ \mathsf{a}_{env}\ (a :: ls)\ cnt \ast rk \mapsto cnt, env'\}$

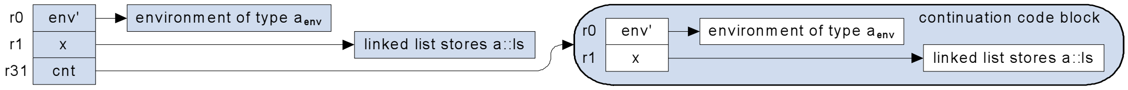
```
        ld r31, r3(0)
```

$\{\langle \cdots\cdots\cdots \wedge \mathtt{r}_{31} = cnt \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \cdots\cdots\cdots \ast rk \mapsto \_, env'\}$

```
        ld r0, r3(1)
```

$\{\langle \mathtt{r}_0 = env' \wedge \cdots\cdots\cdots \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \cdots\cdots\cdots \ast rk \mapsto \_, \_\}$

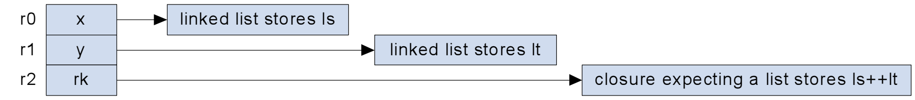
```
        free r3, 2
```



$\{\langle \mathtt{r}_0 = env' \wedge \mathtt{r}_1 = x \wedge \mathtt{r}_{31} = cnt \rangle \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \mathsf{list}\ (a :: ls)\ x \ast \mathsf{a}_{env}\ env' \mathrel{\wedge\!\!\!\wedge\!\!\!\wedge} \mathsf{cont}\ \mathsf{a}_{env}\ (a :: ls)\ cnt\}$

```
        jmp r31
```

Figure 4.2: Code, specification, and illustration of the list append function



$\{\langle \mathtt{r}_0 = x \wedge \mathtt{r}_1 = y \wedge \mathtt{r}_2 = rk \rangle \mathrel{\wedge\!\!\wedge} \mathsf{list}\ ls\ x * \mathsf{list}\ lt\ y * \mathsf{clos}\ (ls +\!\!+ lt)\ rk\}$

```
// the following ‘‘then’’ branch is almost same as the second half of function k
append: bgti r0, 0, else
```

$\{\langle \mathtt{r}_1 = y \wedge \mathtt{r}_2 = rk \rangle \mathrel{\wedge\!\!\wedge} \mathsf{list}\ lt\ y * \mathsf{a}_{env}\ env \mathrel{\wedge\!\!\wedge} \mathsf{cont}\ \mathsf{a}_{env}\ lt\ cnt * rk \mapsto cnt, env\}$

```
        ld r31, r2(0)
```

$\{\langle \cdots\cdots \wedge \mathtt{r}_{31} = cnt \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * rk \mapsto \_, env\}$

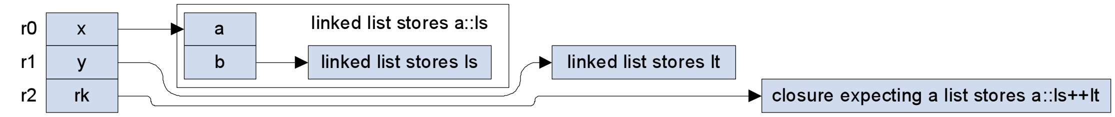
```
        ld r0, r2(1)
```

$\{\langle \mathtt{r}_0 = env \wedge \cdots\cdots \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * rk \mapsto \_, \_\}$

```
        free r2, 2
```

$\{\langle \mathtt{r}_0 = env \wedge \mathtt{r}_1 = y \wedge \mathtt{r}_{31} = cnt \rangle \mathrel{\wedge\!\!\wedge} \mathsf{list}\ lt\ y * \mathsf{a}_{env}\ env \mathrel{\wedge\!\!\wedge} \mathsf{cont}\ \mathsf{a}_{env}\ lt\ cnt\}$

```
        jmp r31
```



$\{\langle \mathtt{r}_0 = x \wedge \mathtt{r}_1 = y \wedge \mathtt{r}_2 = rk \rangle \mathrel{\wedge\!\!\wedge} \mathsf{list}\ ls\ b * \mathsf{list}\ lt\ y * \mathsf{clos}\ (a :: ls +\!\!+ lt)\ rk * x \mapsto a, b\}$

```
else:   alloc r3, 2
```

$\{\langle \cdots\cdots \wedge \mathtt{r}_3 = env \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * env \mapsto \_, \_\}$

```
        st r3(0), r0
```

$\{\langle \cdots\cdots \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * env \mapsto x, \_\}$

```
        st r3(1), r2
```

$\{\langle \cdots\cdots \wedge \cdots\cdots \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * env \mapsto x, rk\}$

```
        ld r0, r0(1)
```

$\{\langle \mathtt{r}_0 = b \wedge \cdots\cdots \wedge \cdots\cdots \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * x \mapsto a, \_ * \cdots\cdots\}$

```
        alloc r2, 2
```

$\{\langle \cdots\cdots \wedge \mathtt{r}_2 = nk \wedge \cdots\cdots \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * nk \mapsto \_, \_\}$

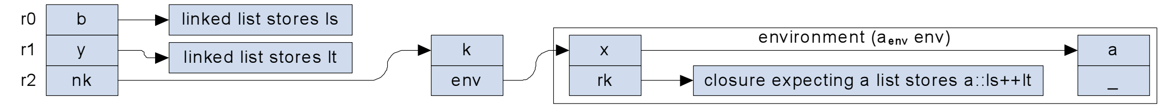
```
        st r2(1), r3
```

$\{\langle \cdots\cdots \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots * nk \mapsto \_, env\}$

```
        movi r3, k
```

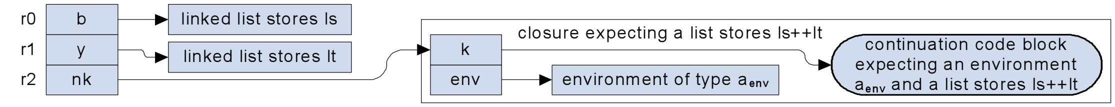
$\{\langle \cdots\cdots \wedge \mathtt{r}_3 = k \rangle \mathrel{\wedge\!\!\wedge} \cdots\cdots\}$

```
        st r2(0), r3
```



$\{\langle \mathtt{r}_0 = b \wedge \mathtt{r}_1 = y \wedge \mathtt{r}_2 = nk \rangle \mathrel{\wedge\!\!\wedge} \mathsf{list}\ ls\ b * \mathsf{list}\ lt\ y * \mathsf{clos}\ (a :: ls +\!\!+ lt)\ rk * x \mapsto a, \_ * env \mapsto x, rk * nk \mapsto k, env\}$

```
        // (ecp) rule
```


r0
r1
r2
b
y
nk
linked list stores ls
linked list stores lt
closure expecting a list stores ls++lt
k
env
environment of type aenv
continuation code block expecting an environment aenv and a list stores ls++lt

$\{\langle \mathtt{r}_0 = b \wedge \mathtt{r}_1 = y \wedge \mathtt{r}_2 = nk \rangle \mathrel{\wedge\!\!\wedge} \mathsf{list}\ ls\ b * \mathsf{list}\ lt\ y * \mathsf{clos}\ (ls +\!\!+ lt)\ nk\}$

// *where* a_{env} *being packed is defined as* $\mathsf{a}_{env}\ env \triangleq \mathsf{clos}\ (a :: ls +\!\!+ lt)\ rk * x \mapsto a, \_ * env \mapsto x, rk$

```
        jd append
```

Figure 4.3: Code, specification, and illustration of the list append function (continued)

Our specification of the *append* function guarantees that the return continuation *rk* will get a **correctly** appended list (*i.e.*, the list contents are exactly the same as the two input lists). Furthermore, even though the function contains a large amount of heap allocation, mutation, and deallocation (which are used to build and destroy continuation closures and to append two lists on the fly), memory safety is **fully** guaranteed (no garbage, no illegal free operation).

4.3 Recursive Specifications

Recursive specifications are very useful in describing complex invariants. Simple recursive data structures such as link-list are already supported by the inductive definition found in *Prop*. However, for the recursive types ($\mu\alpha.\tau$) found in TAL, where things such as embedded code pointers can be nested inside an inductive definition, their counterpart in logic, "recursive predicates", can not be easily defined in XCAP. We extend *PropX* to support recursive predicates and use syntactic methods to establish validity.

We define the recursive predicate constructor as follows.

$$(\mathit{PropX}) \quad \mathtt{P},\mathtt{Q} \quad ::= \quad \ldots \mid (\boldsymbol{\mu}\,\alpha{:}A \rightarrow \mathit{PropX}.\,\lambda x{:}A.\mathtt{P}\ \ B)$$

To understand the formation of recursive predicate, we will start from the innermost proposition $\mathtt{P}$. $\lambda x{:}A.\mathtt{P}$ is a predicate of type $A \rightarrow \mathit{PropX}$. So $\boldsymbol{\mu}\alpha{:}A \rightarrow \mathit{PropX}.\lambda x{:}A.\mathtt{P}$ is meant to be a recursive predicate of type $A \rightarrow \mathit{PropX}$, which corresponds to recursive types found in type systems. Since the basic unit of definition is extended proposition instead of extended predicate, we apply it with a term B of type A, and make $(\boldsymbol{\mu}\,\alpha{:}A \rightarrow \mathit{PropX}.\,\lambda x{:}A.\mathtt{P}\ \ B)$ the basic shape of recursive predicates formula. When using recursive predicates formulas, we often use its predicate form, and use the notation $\boldsymbol{\mu}\alpha{:}A \rightarrow \mathit{PropX}.\lambda x{:}A.\mathtt{P}$ to represent predicate $\lambda y{:}A.(\boldsymbol{\mu}\,\alpha{:}A \rightarrow \mathit{PropX}.\,\lambda x{:}A.\mathtt{P}\ \ y)$.

To establish validity of recursive predicate formulas, we add the following rule to the validity rules in Figure 4.1. It is essentially a fold rule for recursive types.

$$\frac{B\!:\!A \qquad \Gamma \vdash \mathtt{P}[B/x][\boldsymbol{\mu}\alpha\!:\!A \to \mathit{PropX}.\,\lambda x\!:\!A.\,\mathtt{P}/\alpha]}{\Gamma \vdash (\boldsymbol{\mu}\,\alpha\!:\!A \to \mathit{PropX}.\,\lambda x\!:\!A.\,\mathtt{P}\ \ B)}\ (\mu\text{-}\mathrm{I})$$

We extend *PropX* interpretation soundness (Theorem 4.1) with the following case:

$$\text{If } \llbracket (\boldsymbol{\mu}\,\alpha\!:\!A \to \mathit{PropX}.\,\lambda x\!:\!A.\,\mathtt{P}\ \ B) \rrbracket_\Psi \text{ then } \llbracket \mathtt{P}[B/x][\boldsymbol{\mu}\alpha\!:\!A \to \mathit{PropX}.\,\lambda x\!:\!A.\,\mathtt{P}/\alpha] \rrbracket_\Psi.$$

4.4 Discussion

In Figure 4.1 we have only included the introduction rules for the two impredicative quantifiers. This could cause confusion because from the logic perspective, missing the two elimination rules would raise questions related to the completeness of the logic. However, despite its name, *PropX* is **not** designed to be a general (complete) logic; it is purely a level of syntax laid upon the meta logic. While its expressive power comes from the lifted propositions $\langle p \rangle$, the modular handling of ECPs and impredicative polymorphism follows syntactic types.

To certify the examples in this dissertation (or any polymorphic TAL programs), what we need is to establish the assertion subsumption relation $\Rrightarrow$ between XCAP assertions. According to its definition, assertion subsumption is merely a meta-implication between validities of XCAP propositions. Although in certain cases it is possible to first do all the subsumption reasoning in *PropX* and prove $\llbracket \mathtt{P} \twoheadrightarrow \mathtt{Q} \rrbracket_\Psi$, and then obtain the subsumption proof $\llbracket \mathtt{P} \rrbracket_\Psi \supset \llbracket \mathtt{Q} \rrbracket_\Psi$ by Theorem 4.1, it is not always possible due to the lack of completeness for *PropX*, and is not the way *PropX* should be used. Instead, one can always follow the diagram below in proving subsumption relations (we use the impredicative existential quantifier as an example):

$$\begin{tikzcd}
\llbracket \exists\alpha.\,\mathtt{P} \rrbracket_\Psi \arrow[r, dashed, "\textit{implication}"] \arrow[d, Rightarrow, "\textit{Theorem 4.1}"] & \llbracket \exists\alpha.\,\mathtt{Q} \rrbracket_\Psi \\
\exists \mathtt{a}.\,\llbracket \mathtt{P}[\mathtt{a}/\alpha] \rrbracket_\Psi \arrow[r, "\textit{meta-implication}"] & \exists \mathtt{a}.\,\llbracket \mathtt{Q}[\mathtt{a}/\alpha] \rrbracket_\Psi \arrow[u, Rightarrow, "\text{rule } \exists\text{-I2}"']
\end{tikzcd}$$

To prove the intended "implication" relation (the top one), we first use Theorem 4.1 to turn the source proposition's existential quantification into the meta one, from which we can do (flexible) meta implications. Then we reconstruct the existential quantification of the target proposition via the introduction rule. This way, the construction of subsumption proof in meta logic does not require the reasoning at the *PropX* level.

In fact, the subtyping relation found in TAL can be simulated by the subsumption relation in XCAP (with only the introduction rules for the two impredicative quantifiers). What the missing "elimination rules" would add is the ability to support a notion of "higher-order subtyping" between "impredicative types", which does not appear in practical type systems such as TAL, FLINT, or ML. Although it could be nice to include such a feature in XCAP, we did not do so since that would require a complex semantic normalization proof instead of the simple syntactic one used for Theorem 4.1.

As we will show in the next Chapter, this is enough for reasoning about impredicative polymorphism available in typical type systems. Translations from polymorphic type systems such as TAL to XCAP also do not require the elimination rules.

Similarly explanation applies to recursive specifications, for which only the introduction rules are defined, too.

Chapter 5

Weak Updates and a Translation from Typed Assembly Language

Weak update, also termed as "general reference", is another higher-order features that logic-based verification methods failed to support well. In this chapter we first show how to extend the XCAP framework to support weak update, using similar syntactic technique for the support of ECP in Chapter 3.

We then explore the relationship between XCAP and typed assembly languages (TAL). TAL and CAP/XCAP are suitable for different kinds of verification tasks. Previously, programs verified in either one of them can not interoperate freely with the other, making it hard to integrate them into a complete system. Moreover, the relationship between TAL and CAP lines of work has not been discussed extensively.

In this chapter, we compare the type-based and logic-based methods by presenting a type-preserving translation from a TAL language to XCAP. The translation involves an intermediate step of a "semantic" TAL language. Our translation supports polymorphic code, mutable reference, existential, and recursive types. Since we proved typing preservation for the translation from TAL to XCAP, there is a clear path to link and interoperate well-typed programs from traditional certifying compilers with certified libraries by CAP-like systems.

5.1 Weak Update in the Logical Setting

Weak update (also termed as "mutable reference" or "general references") is a commonly used memory mutation model. Examples of weak update include ML reference cells (int ref) and managed data pointers (int __gc*) in .NET common type system. In the weak update model, the value of each memory cell must satisfy a certain fixed value type. The tuple type in TAL (such as the one to be shown in next section) is also a weak update reference cell types.

Unfortunately, weak update is not well-supported by CAP and other Hoare-logic-based verification systems. The problem is due to the lack of a global data heap invariant that local assertions about heap cells can be checked upon. Existing Hoare-logic-based systems either avoid supporting weak update [44, 64], limit the assertions (for reference cells) to types only [28], or resort to heavyweight techniques that require the construction of complex semantic models [9, 2]. In this section, we present a weak update extension of the XCAP using the similar syntactic technique for ECPs.

Following cptr for ECP, We add a reference cell proposition $\mathsf{ref}(\mathtt{l},\mathtt{t})$ to the extended propositions. It associates word type $\mathtt{t}$ (a value predicate) with data label $\mathtt{l}$.

$$\begin{array}{lccl} (\mathit{PropX}) & \mathtt{P},\mathtt{Q} & ::= & \ldots \mid \mathsf{ref}(\mathtt{l},\mathtt{t}) \\ (\mathit{WordTy}) & \mathtt{t} & \in & \mathit{Word} \rightarrow \mathit{PropX} \end{array}$$

We can use the following macro to describe a record of n cells.

$$\mathsf{record}(\mathtt{l},\mathtt{t}_1,\ldots,\mathtt{t}_n) \triangleq \mathsf{ref}(\mathtt{l},\mathtt{t}_1) \mathbin{\&\!\&} \ldots \mathbin{\&\!\&} \mathsf{ref}(\mathtt{l}{+}n{-}1,\mathtt{t}_n)$$

To testify the validity of reference cell propositions, in the interpretation $[\![\mathtt{P}]\!]_{\Psi,\Phi}$ we need an additional "data heap specification" parameter Φ which, similar to the code heap specification Ψ, is a partial mapping from data labels to word types.

$$(\mathit{DtHpSpec}) \qquad \Phi \quad ::= \quad \{\mathtt{l} \rightsquigarrow \mathtt{t}\}^*$$

To establish validity of $\mathsf{ref}(\mathtt{l},\mathtt{t})$, data label $\mathtt{l}$ and word type $\mathtt{t}$ need to be in Φ.

$$\frac{\Phi(\mathtt{l}) = \mathtt{t}}{\Gamma \vdash_{\Psi,\Phi} \mathsf{ref}(\mathtt{l},\mathtt{t})}\ (\textsc{RF-I}) \qquad \frac{\Gamma \vdash_{\Psi,\Phi} \mathsf{ref}(\mathtt{l},\mathtt{t}) \qquad (\Phi(\mathtt{l}) = \mathtt{t}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash_{\Psi,\Phi} \mathtt{Q}}\ (\textsc{RF-E})$$

Validity rules of other cases remain unchanged other than taking an extra Φ argument. Validity soundness of those cases also holds, with the following additional case.

$$\text{If } [\![\, \mathsf{ref}(\mathtt{l},\mathtt{t}) \,]\!]_{\Psi,\Phi} \text{ then } \Phi(\mathtt{l}) = \mathtt{t};$$

Different from code heap, the data heap is dynamic. Its specification can not be obtained statically. Instead, we need to find it out in each execution steps. So the assertion interpretation is changed to:

$$[\![\, \mathtt{a} \,]\!]_{\Psi} \triangleq \lambda(\mathbb{H},\mathbb{R}).\, \exists \Phi, \mathbb{H}_s, \mathbb{H}_w.\, \mathbb{H} = \mathbb{H}_s \uplus \mathbb{H}_w \wedge [\![\, \mathtt{a}\ (\mathbb{H}_s,\mathbb{R}) \,]\!]_{\Psi,\Phi} \wedge \mathcal{DH}\ \Psi\ \Phi\ \mathbb{H}_w$$

There should exist a data heap specification Φ describing the weak update part of current data heap. Other than checking validity of $(\mathtt{a}\ (\mathbb{H},\mathbb{R}))$ using Ψ and Φ, we also need to checking validity of Φ.

For a data heap specification Φ to be valid, each reference cell must contain a value that matches its word type.

$$\mathcal{DH}\ \Psi\ \Phi\ \mathbb{H} \ \triangleq\ \forall \mathtt{l} \in \mathsf{dom}(\Phi) = \mathsf{dom}(\mathbb{H}).\, [\![\, \Phi(\mathtt{l})\ \mathbb{H}(\mathtt{l}) \,]\!]_{\Psi,\Phi}$$

The extension of XCAP to support weak update requires minor changes to the assertion languages and interpretations, and zero change to the inference rules. Thus the soundness of XCAP is easily preserved.

Based on the weak update memory model defined above, we can derive many useful "macro" inference rules as shown below. These rules can help guide the proof process for the programmer. ($\mathsf{insens}(\mathtt{a},\mathtt{r})$ asserts predicate is insensitive to register $\mathtt{r}$, *i.e.*, does not talk about register $\mathtt{r}$. Its definition is omitted here.)

$$\frac{\Psi \vdash \{(\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}(\mathbb{H},\mathbb{R}) \Wedge \mathtt{t}\ \mathbb{R}(\mathtt{r}_d))\}\, \mathbb{I} \qquad \mathsf{insens}(\mathtt{a},\mathtt{r}_d)}{\Psi \vdash \{(\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}(\mathbb{H},\mathbb{R}) \Wedge \mathsf{ref}(\mathbb{R}(\mathtt{r}_s) + \mathtt{w}, \mathtt{t}))\}\, \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w}); \mathbb{I}}\ (\textsc{W-LD})$$

$$\frac{\Psi \vdash \{\mathtt{a}\}\, \mathbb{I}}{\Psi \vdash \{(\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}(\mathbb{H},\mathbb{R}) \Wedge \mathsf{ref}(\mathbb{R}(\mathtt{r}_d) + \mathtt{w}, \mathtt{t}) \Wedge \mathtt{t}\ \mathbb{R}(\mathtt{r}_s))\}\, \mathsf{st}\ \mathtt{r}_d(\mathtt{w}), \mathtt{r}_s; \mathbb{I}}\ (\textsc{W-ST})$$

Example. Below is a "mini object", using the recursive predicates and weak update extensions presented in previous sections.

```
classs c {
  void f (c x) { x.f(x) }
}
```

$$c \ \triangleq\ \boldsymbol{\mu}\ \alpha{:}\mathit{Word}\rightarrow\mathit{PropX}.\lambda x{:}\mathit{Word}.\mathsf{ref}(x,\ \lambda y{:}\mathit{Word}.\mathsf{cptr}(y,\ \lambda(\mathbb{H},\mathbb{R}).(\alpha\ \mathbb{R}(\mathtt{r1}))))$$

The above example may look similar to what one would normally write in a syntactic type system. However, given the ability to write general logic predicate in the specifications, it is possible to compose interesting data structures and properties. Below is an example: a record pointer `l` which points to an even number, a data pointer to a reference cell storing an odd number, and a code pointer which, among other things, expects register $\mathtt{r}_1$ to be an unaliased pointer to a prime number. (here we used the separation logic primitives embedded in Coq, as discussed in Section 4.2).

$$\mathsf{record}(\mathtt{l},\ \mathsf{even},\ \lambda\mathtt{w}.\mathsf{ref}(\mathtt{w},\mathsf{odd}),\ \lambda\mathtt{w}.\mathsf{cptr}(\mathtt{w},\ \lambda(\mathbb{H},\mathbb{R}).(\exists\mathtt{w}.\mathbb{R}(\mathtt{r}_1)\mapsto\mathtt{w}\wedge\mathsf{prime}\ \mathtt{w})*\ldots))$$

5.2 Typed Assembly Language (TAL)

The TAL language presented here follows the principle of the original typed assembly languages [41]. However, due to the usage of TM, a untyped raw machine, the shape of typing judgments and rules are slightly different. To simplify our presentation, the TAL in this chapter does not directly deal with heap allocation/deallocation.

Type definitions. Figure 5.1 presents the type definitions in TAL. Machine word is classified as of value type (τ) including integer, code, tuple, existential package, and recursive data structures. A code heap specification (Ψ) is a partial environment that maps a code label to a "precondition" type ($[\Delta].\Gamma$) for its corresponding code block. Here Δ is a type variable environment and Γ is a register file type which specifies the type for each register. Similarly, a data heap specification (Φ) is a partial environment that maps from a data label to a value type for its corresponding heap cell.

$$\begin{array}{rl}
(\mathit{CdHpSpec}) & \Psi ::= \{\mathtt{f} \rightsquigarrow [\Delta].\Gamma\}^{*} \\
(\mathit{RfileTy}) & \Gamma ::= \{\mathtt{r} \rightsquigarrow \tau\}^{*} \\
(\mathit{TyVarEnv}) & \Delta ::= \cdot \mid \alpha, \Delta \\
(\mathit{WordTy}) & \tau ::= \alpha \mid \mathsf{int} \mid \mathsf{code}\ [\Delta].\Gamma \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \exists \alpha.\, \tau \mid \mu\alpha.\, \tau \\
(\mathit{DtHpSpec}) & \Phi ::= \{\mathtt{l} \rightsquigarrow \tau\}^{*}
\end{array}$$

Figure 5.1: Type definitions of TAL

Static semantics. The top-level semantic rules of TAL are presented in Figure 5.2. A program is well-formed if each of its components is. For a code heap to be well-formed, each block in it must be well-formed. The intuition behind well-formed instruction sequence judgment is that if the state satisfies the precondition $[\Delta].\Gamma$, then executing $\mathbb{I}$ is safe with respect to Ψ. Weakening is allowed to turn one precondition into another provided that they satisfy the subtyping relation. A instruction sequence $\mathtt{c};\mathbb{I}$ is safe if one can find another register file type which serves as both the post-condition of $\mathtt{c}$ and the precondition of $\mathbb{I}$. A direct jump is safe if the current precondition implies the precondition of the target code block specified in Ψ. An indirect jump is safe when the target register is of a code type with a weaker precondition. Constant code labels can be moved into registers.

The subtyping and instruction typing rules for TAL is presented in Figure 5.3. Valid subtypings include dropping registers, instantiation of code type, packing and unpacking of existential packages, and folding and unfolding of recursive types. It allows jumping to code blocks with stronger preconditions than required.

For each non-control-flow-transfer instruction there is a pre/post-condition relation defined. When a register is updated by an instruction, the register file type is also updated by a new value type. Arithmetic instruction only operates on two registers with integer types. Since there is no static data heap, a constant value can only be well-formed under an empty Heap type before it can be moved to a register. For simple instructions, their pre- and post-conditions do not involve a change of type variable environment, thus we only specify the register file types before and after their execution. The instruction-level

$\boxed{\Psi_G \vdash \{[\Delta].\Gamma\}\,\mathbb{P}}$ **(*Well-formed Program*)**

$$\frac{\Psi_G \vdash \mathbb{C} : \Psi_G \qquad \Psi_G \vdash \mathbb{S} : [\Delta].\Gamma \qquad \Psi_G \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}{\Psi_G \vdash \{[\Delta].\Gamma\}\,(\mathbb{C},\mathbb{S},\mathbb{I})}\ (\textsc{prog})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C} : \Psi}$ **(*Well-formed Code Heap*)**

$$\frac{\Psi_{IN} \vdash \{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f}) \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi_{IN} \vdash \mathbb{C} : \Psi}\ (\textsc{cdhp})$$

$$\frac{\begin{array}{l}\Psi_{IN\,1} \vdash \mathbb{C}_1 : \Psi_1 \qquad \Psi_{IN\,2} \vdash \mathbb{C}_2 : \Psi_2 \qquad \Psi_{IN\,1}(\mathtt{f}) = \Psi_{IN\,2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \qquad\qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN\,1}) \cap \mathsf{dom}(\Psi_{IN\,2})\end{array}}{\Psi_{IN\,1} \cup \Psi_{IN\,2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2}\ (\textsc{link})$$

$\boxed{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}$ **(*Well-formed Instruction Sequence*)**

$$\frac{\vdash \{\Gamma\}\,\mathsf{c}\,\{\Gamma'\} \qquad \Psi \vdash \{[\Delta].\Gamma'\}\,\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{mov}, \mathsf{movi}, \mathsf{ld}, \mathsf{st}\}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{c};\mathbb{I}}\ (\textsc{seq})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \vdash [\Delta].\Gamma \le \Psi(\mathtt{f})}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{jd}\ \mathtt{f}}\ (\textsc{jd})$$

$$\frac{\Gamma(\mathtt{r}) = \mathsf{code}\ [\Delta'].\Gamma' \qquad \vdash [\Delta].\Gamma \le [\Delta'].\Gamma'}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Gamma(\mathtt{r}_s) = \mathsf{int} \qquad \Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I} \qquad \vdash [\Delta].\Gamma \le \Psi(\mathtt{f})}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f};\mathbb{I}}\ (\textsc{bgti})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{[\Delta].\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{code}\ \ \Psi(\mathtt{f})\}\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{movi}\ \mathtt{r}_d, \mathtt{f};\mathbb{I}}\ (\textsc{movf})$$

$$\frac{\vdash [\Delta].\Gamma \le [\Delta'].\Gamma' \qquad \Psi \vdash \{[\Delta'].\Gamma'\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}\ (\textsc{weaken})$$

Figure 5.2: Top-level static semantics of TAL

$\boxed{\vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'}$ ***(Subtyping)***

$$\frac{\Delta \supseteq \Delta' \qquad \forall\, \texttt{r} \in \mathsf{dom}(\Gamma') \qquad \Gamma(\texttt{r}) = \Gamma'(\texttt{r}) \qquad \Delta' \vdash \Gamma'(\texttt{r})}{\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'}\ (\textsc{subt})$$

$$\frac{\Gamma(\texttt{r}) = \mathsf{code}\ [\alpha, \Delta'].\,\Gamma' \qquad \Delta \vdash \tau'}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\texttt{r} : \mathsf{code}\ [\Delta'].\,\Gamma'[\tau'/\alpha]\}}\ (\textsc{tapp})$$

$$\frac{\Gamma(\texttt{r}) = \tau[\tau'/\alpha] \qquad \Delta \vdash \tau'}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\texttt{r} : \exists\alpha.\,\tau\}}\ (\textsc{pack})$$

$$\frac{\Gamma(\texttt{r}) = \exists\alpha.\,\tau}{\vdash [\Delta].\,\Gamma \leq [\alpha, \Delta].\Gamma\{\texttt{r} : \tau\}}\ (\textsc{unpack})$$

$$\frac{\Gamma(\texttt{r}) = \tau[\mu\alpha.\,\tau/\alpha]}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\texttt{r} : \mu\alpha.\,\tau\}}\ (\textsc{fold})$$

$$\frac{\Gamma(\texttt{r}) = \mu\alpha.\,\tau}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\texttt{r} : \tau[\mu\alpha.\,\tau/\alpha]\}}\ (\textsc{unfold})$$

$\boxed{\vdash \{\Gamma\}\ \mathsf{c}\ \{\Gamma'\}}$ ***(Well-formed Instruction)***

$$\frac{\Gamma(\texttt{r}_s) = \Gamma(\texttt{r}_t) = \mathsf{int}}{\vdash \{\Gamma\}\ \mathsf{add}\ \texttt{r}_d, \texttt{r}_s \texttt{r}_t\ \{\Gamma\{\texttt{r}_d : \mathsf{int}\}\}}\ (\textsc{add})$$

$$\frac{\Gamma(\texttt{r}_s) = \mathsf{int}}{\vdash \{\Gamma\}\ \mathsf{addi}\ \texttt{r}_d, \texttt{r}_s, i\ \{\Gamma\{\texttt{r}_d : \mathsf{int}\}\}}\ (\textsc{addi})$$

$$\frac{\Gamma(\texttt{r}_s) = \tau}{\vdash \{\Gamma\}\ \mathsf{mov}\ \texttt{r}_d, \texttt{r}_s\ \{\Gamma\{\texttt{r}_d : \tau\}\}}\ (\textsc{mov})$$

$$\frac{}{\vdash \{\Gamma\}\ \mathsf{movi}\ \texttt{r}_d, \texttt{w}\ \{\Gamma\{\texttt{r}_d : \mathsf{int}\}\}}\ (\textsc{movi})$$

$$\frac{\Gamma(\texttt{r}_s) = \langle \tau_1, \ldots, \tau_{\texttt{w}+1}, \ldots, \tau_n \rangle}{\vdash \{\Gamma\}\ \mathsf{ld}\ \texttt{r}_d, \texttt{r}_s(\texttt{w})\ \{\Gamma\{\texttt{r}_d : \tau_{\texttt{w}+1}\}\}}\ (\textsc{ld})$$

$$\frac{\Gamma(\texttt{r}_d) = \langle \tau_1, \ldots, \tau_{\texttt{w}+1}, \ldots, \tau_n \rangle \qquad \Gamma(\texttt{r}_s) = \tau_{\texttt{w}+1}}{\vdash \{\Gamma\}\ \mathsf{st}\ \texttt{r}_d(\texttt{w}), \texttt{r}_s\ \{\Gamma\}}\ (\textsc{st})$$

Figure 5.3: Static semantics of TAL (subtyping and instruction typing)

typing rules is not very flexible and expressive, *e.g.*, pointer arithmetic is not possible.

The typing rules for value types, machine state, register file, data heap, and machine word values are presented in Figure 5.4. A value type is well-formed only when it contains no free type variable and, thus, is a ground type. The well-formed state rule instantiates the current type variable environment and requires a current data heap specification to be supplied and checked. State typing is done by checking the Heap and register file's well-formednesses separately under this heap specification. Heap and register file typing further break heap and register file into single word values, and check their well-formedness individually. Any machine word value can be typed as an integer. A label can be typed as a tuple pointer if the cell types in the heap type starting from the label match the corresponding value types in the tuple type. For a label to be considered a code pointer, the code precondition in it has to match the precondition listed in the global context.

Soundness. The soundness theorem guarantees that given a well-formed program, the machine will never get stuck. It is proved following the syntactic approach of proving type soundness [59]. We also list a few key lemmas below.

Theorem 5.1 (TAL Soundness)

If $\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{P}$, for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

Lemma 5.2 (TAL State Weakening)

If $\Psi \vdash \mathbb{S} : [\Delta].\Gamma$ and $\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'$ then $\Psi \vdash \mathbb{S} : [\Delta'].\Gamma'$.

Lemma 5.3 (TAL Instruction Typing)

If $\Psi \vdash \mathbb{S} : [\Delta].\Gamma$ and $\vdash \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\Psi \vdash \mathtt{Next}_{\mathtt{c}}(\mathbb{S}) : [\Delta].\Gamma'$.

5.3 A "Semantic" TAL Language

Instead of doing translation from TAL to XCAP directly, we create a new "Semantic" TAL language (STAL) to serve as an intermediate step between them. Let us revisit the static

$$\boxed{\Delta \vdash \tau \qquad \Psi \vdash \mathbb{S} : [\Delta].\,\Gamma \qquad \Psi \vdash \mathbb{H} : \Phi \qquad \Psi;\Phi \vdash \mathbb{R} : \Gamma}$$

(*Well-formed Type, State, Heap, and Register file*)

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash \tau}\ (\textsc{type})$$

$$\frac{\cdot \vdash \tau_i \qquad \forall\, i \qquad \Psi \vdash \mathbb{H} : \Phi \qquad \Psi;\Phi \vdash \mathbb{R} : \Gamma[\tau_1, \ldots, \tau_n/\alpha_1, \ldots, \alpha_n]}{\Psi \vdash (\mathbb{H}, \mathbb{R}) : [\alpha_1, \ldots, \alpha_n].\Gamma}\ (\textsc{state})$$

$$\frac{\Psi;\Phi \vdash \mathbb{H}(\mathtt{l}) : \Phi(\mathtt{l}) \qquad \forall\, \mathtt{l} \in \mathsf{dom}(\Phi) = \mathsf{dom}(\mathbb{H})}{\Psi \vdash \mathbb{H} : \Phi}\ (\textsc{heap})$$

$$\frac{\Psi;\Phi \vdash \mathbb{R}(\mathtt{r}) : \Gamma(\mathtt{r}) \qquad \forall\, \mathtt{r} \in \mathsf{dom}(\Gamma)}{\Psi;\Phi \vdash \mathbb{R} : \Gamma}\ (\textsc{rfile})$$

$\boxed{\Psi;\Phi \vdash \mathtt{w} : \tau}$ **(*Well-formed Word Value*)**

$$\frac{}{\Psi;\Phi \vdash \mathtt{w} : \mathsf{int}}\ (\textsc{int})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi;\Phi \vdash \mathtt{f} : \mathsf{code}\ \ \Psi(\mathtt{f})}\ (\textsc{code})$$

$$\frac{\cdot \vdash \tau' \qquad \Psi;\Phi \vdash \mathtt{f} : \mathsf{code}\ [\alpha, \Delta].\,\Gamma}{\Psi;\Phi \vdash \mathtt{f} : \mathsf{code}\ [\Delta].\,\Gamma[\tau'/\alpha]}\ (\textsc{poly})$$

$$\frac{\Phi(\mathtt{l}+i-1) = \tau_i \qquad \forall\, i}{\Psi;\Phi \vdash \mathtt{l} : \langle \tau_1, \ldots, \tau_n \rangle}\ (\textsc{tup})$$

$$\frac{\cdot \vdash \tau' \qquad \Psi;\Phi \vdash \mathtt{w} : \tau[\tau'/\alpha]}{\Psi;\Phi \vdash \mathtt{w} : \exists \alpha.\,\tau}\ (\textsc{ext})$$

$$\frac{\Psi;\Phi \vdash \mathtt{w} : \tau[\mu\alpha.\,\tau/\alpha]}{\Psi;\Phi \vdash \mathtt{w} : \mu\alpha.\,\tau}\ (\textsc{rec})$$

Figure 5.4: Static semantics of TAL (state and value typing)

semantics of TAL in Figure 5.3 and the lemmas used in TAL soundness proof.

First, look at the set of subtyping rules between preconditions ($\vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'$). If we also look at the State Weakening lemma (Lemma 5.2) in TAL soundness proof, it is easy to see that all of those syntactic rules are just used for the meta implication between the two state typings in the soundness proof. Given a mechanized meta logic, we can replace these rules with a single one,

$$\frac{\Psi \vdash \mathbb{S} : [\Delta].\,\Gamma \;\supset\; \Psi \vdash \mathbb{S} : [\Delta'].\,\Gamma' \qquad \forall\, \mathbb{S}, \Psi}{\vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'} \;(\textsc{subt})$$

which explicitly requests the meta implications between two state typing. By doing so, not only did we remove the fixed syntactic subtyping rules from TAL, but the State Weakening lemma is no longer part of the soundness proof. More importantly, the subtyping between preconditions is no longer limited to the built-in rules. Any valid implication relation between state typing is allowed. This is much more flexible and powerful.

Now let us look at the set of instruction typing rules ($\vdash \{\Gamma\}\mathtt{c}\{\Gamma'\}$). Also look at the Instruction Typing lemma (Lemma 5.3) in TAL soundness proof. Again all these syntactic rules are just used for the meta implication between two state typings in the soundness proof. (The difference this time is that the two states are now different and are states before and after the execution of an instruction, respectively.) Applying our trick again, we can replace these rules with a single one,

$$\frac{\Psi \vdash \mathbb{S} : [\Delta].\,\Gamma \;\supset\; \Psi \vdash \mathtt{Next}_{\mathtt{c}}(\mathbb{S}) : [\Delta].\,\Gamma' \qquad \forall\, \mathbb{S}, \Psi}{\vdash \{\Gamma\}\mathtt{c}\{\Gamma'\}} \;(\textsc{instr}).$$

We also successfully removed the fixed syntactic instruction typing rules as well as he Instruction Typing lemma from TAL. The new form of instruction typing is much more flexible and powerful.

Of course, in composing the actual meta proof supplied to the new SUBT and INSTR rules, it is most likely that those disappeared lemmas will still be used. Nevertheless, making them separate from the type language and its meta theory is important because it reduces the size of the type language, while allowing more flexible reasoning. By introducing meta implication into TAL, we made one step forward so now there is a mixture of syntactic types and logic proofs.

We call this version of TAL a *semantic* TAL (STAL). STAL and TAL share the exactly same syntax and top-level static semantics (Figure 5.2), as well as the same state and value typing rules (Figure 5.4). While STAL has much more reasoning power than TAL do, as the soundness of STAL is simpler than TAL's. Nevertheless, instead of merely supplying type signatures to their code, now the programmers have to also supply meta logic proof.

As an intermediate step from TAL toward XCAP, STAL only upgrades TAL's expressiveness by using general logic implications in the subtyping and instruction typing rules. STAL program specifications are still fully syntactic types and thus are not as expressive as general logic predicate.

It is obvious to obtain the following TAL-to-STAL typing translation theorem. (We ignore the trivial cases of those judgments where TAL and STAL share the same rules.)

Theorem 5.4 (Typing Preservations from TAL to STAL)

1. if $\vdash_{\text{TAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'$ then $\vdash_{\text{STAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'$;

2. if $\Psi \vdash_{\text{TAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\Psi \vdash_{\text{STAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$.

If we follow the direction of STAL and make one more step by trying to bring in general logic predicates into the types, we can obtain an even better TAL. In fact, XCAP is such a system in some sense, as the translation in the next section will show.

5.4 Translations from TAL/STAL to XCAP

The gap between STAL and XCAP is mainly on the specification language and the various state and value typing. At the core of the translation from STAL to XCAP is the translations from TAL/STAL types and typings into XCAP predicates, which we discuss first. Our translations preserve the typing structure and well-formedness of TM programs.

Translation from TAL WordTy to XCAP WordTy

$$
\begin{aligned}
\ulcorner \mathsf{int} \urcorner &\triangleq \lambda \mathtt{w}.\, \mathsf{True} \\
\ulcorner \mathsf{code}\ [\Delta].\,\Gamma \urcorner &\triangleq \lambda \mathtt{w}.\, \mathsf{codeptr}(\mathtt{w}, \ulcorner [\Delta].\,\Gamma \urcorner) \\
\ulcorner \langle \tau_1, \ldots, \tau_n \rangle \urcorner &\triangleq \lambda \mathtt{w}.\, \mathsf{record}(\mathtt{w}, \ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_n \urcorner) \\
\ulcorner \exists \alpha.\, \tau \urcorner &\triangleq \lambda \mathtt{w}.\, \exists\, \alpha : Word \rightarrow PropX.\, \ulcorner \tau \urcorner\ \mathtt{w} \\
\ulcorner \mu \alpha.\, \tau \urcorner &\triangleq \lambda \mathtt{w}.\, (\boldsymbol{\mu}\, \alpha : Word \rightarrow PropX.\, \lambda x : Word.\, (\ulcorner \tau \urcorner\ x) \quad \mathtt{w})
\end{aligned}
$$

Translation from TAL Precondition to XCAP Assertion

$$
\begin{aligned}
&\ulcorner [\alpha_1, \ldots, \alpha_m].\, \{\mathtt{r}_1 \leadsto \tau_1, \ldots, \mathtt{r}_n \leadsto \tau_n\} \urcorner \\
&\quad \triangleq\ \lambda(\mathbb{H}, \mathbb{R}).\, \exists\, \alpha_1, \ldots, \alpha_m : Word \rightarrow PropX.\, (\ulcorner \tau_1 \urcorner\, \mathbb{R}(\mathtt{r}_1)) \Wedge \ldots \Wedge (\ulcorner \tau_n \urcorner\, \mathbb{R}(\mathtt{r}_n))
\end{aligned}
$$

Translation from TAL CdHpSpec to XCAP CdHpSpec

$$
\ulcorner \{\mathtt{l}_1 \leadsto [\Delta_1].\,\Gamma_1, \ldots, \mathtt{l}_n \leadsto [\Delta_n].\,\Gamma_n\} \urcorner \ \triangleq\ \{\mathtt{l}_1 \leadsto \ulcorner [\Delta_1].\,\Gamma_1 \urcorner, \ldots, \mathtt{l}_n \leadsto \ulcorner [\Delta_n].\,\Gamma_n \urcorner\}
$$

Translation from TAL DtHpSpec to XCAP DtHpSpec

$$
\ulcorner \{\mathtt{l}_1 \leadsto \tau_1, \ldots, \mathtt{l}_n \leadsto \tau_n\} \urcorner \ \triangleq\ \{\mathtt{l}_1 \leadsto \ulcorner \tau_1 \urcorner, \ldots, \mathtt{l}_n \leadsto \ulcorner \tau_n \urcorner\}
$$

Figure 5.5: Translations from TAL types to XCAP predicates

Translation of TAL types We present the type translations from TAL/STAL to XCAP in Figure 5.5. Various $\ulcorner \cdot \urcorner$ translate TAL types into XCAP assertions and specifications.

In the word type translation, integer type becomes a tautology as any machine word can be treated as an integer. Code type is translated into ECP formulas. Tuple types in TAL is translated into record type in XCAP. Existential types and recursive types in TAL are also translated into their XCAP counterparts.

The translation of a TAL precondition, which is a type variable environment plus a register file type, is an XCAP assertion. The type variables in the environment are now existentially quantified over XCAP word types at the outmost of the target assertion. The register file typing corresponds to a bunch of conjunctions of register value testing.

The translations of TAL code and data heap specifications are carried out by simply translating each element's type in them into XCAP assertions or word types, and preserving the partial mapping.

Typing Preservations An important property of the previous translations is whether they preserve the typing structure and well-formedness of TM program in XCAP. This breaks down to whether all well-formed TAL/STAL entities are still well-formed in XCAP after the translations. As the following typing preservation lemma shows, all STAL typing derivations indeed get preserved in XCAP after the translations.

Theorem 5.5 (Typing Preservations from STAL to XCAP)

1. If $\Psi \vdash_{\mathrm{STAL}} \{[\Delta].\Gamma\}\,\mathbb{P}$ then $\ulcorner\Psi\urcorner \vdash_{\mathrm{XCAP}} \{\ulcorner[\Delta].\Gamma\urcorner\}\,\mathbb{P}$;
2. if $\Psi \vdash_{\mathrm{STAL}} \mathbb{C} : \Psi'$ then $\ulcorner\Psi\urcorner \vdash_{\mathrm{XCAP}} \mathbb{C} : \ulcorner\Psi'\urcorner$;
3. if $\Psi \vdash_{\mathrm{STAL}} \{[\Delta].\Gamma\}\mathbb{I}$ then $\ulcorner\Psi\urcorner \vdash_{\mathrm{XCAP}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathbb{I}$;
4. if $\Psi \vdash_{\mathrm{STAL}} \mathbb{S} : [\Delta].\Gamma$ then $[\![\ulcorner[\Delta].\Gamma\urcorner]\!]_{\ulcorner\Psi\urcorner}\,\mathbb{S}$;
5. if $\Psi \vdash_{\mathrm{STAL}} \mathbb{H} : \Phi$ then $\mathcal{DH}\ \ulcorner\Psi\urcorner\ \ulcorner\Phi\urcorner\ \mathbb{H}$;
6. if $\Psi;\Phi \vdash_{\mathrm{STAL}} \mathbb{R} : \Gamma$ then $[\![\ulcorner[\,].\Gamma\urcorner\ (\mathbb{H},\mathbb{R})\,]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$;
7. if $\Psi;\Phi \vdash_{\mathrm{STAL}} \mathtt{w} : \tau$ then $[\![\ulcorner\tau\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$;
8. if $\vdash_{\mathrm{STAL}} [\Delta].\Gamma \le [\Delta'].\Gamma'$ then $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\Gamma'\urcorner$;
9. if $\Psi \vdash_{\mathrm{STAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\ulcorner[\Delta].\Gamma\urcorner_{\Psi} \Rightarrow_{\mathtt{c}} \ulcorner[\Delta].\Gamma'\urcorner$.

Proof. We show selected cases for (3). By induction over the structure of $\Psi \vdash_{\mathrm{TAL}} \{[\Delta].\Gamma\}\mathbb{I}$.

case WEAKEN.

$$\frac{\vdash [\Delta].\Gamma \le [\Delta'].\Gamma' \qquad \Psi \vdash \{[\Delta'].\Gamma'\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}$$

By induction hypodissertation it follows that $\ulcorner\Psi\urcorner \vdash_{\mathrm{XCAP}} \{\ulcorner[\Delta'].\Gamma'\urcorner\}\mathbb{I}$. The following weakening lemma easily holds for XCAP: "if $\Psi \vdash \{\mathtt{a}'\}\,\mathbb{I}$ and $\mathtt{a} \Rightarrow \mathtt{a}'$ then $\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$". By (8) it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\Gamma'\urcorner$. Thus it follows that $\ulcorner\Psi\urcorner \vdash_{\mathrm{XCAP}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathbb{I}$.

case SEQ.

$$\frac{\vdash \{\Gamma\}\,\mathsf{c}\,\{\Gamma'\} \qquad \Psi \vdash \{[\Delta].\,\Gamma'\}\,\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{mov}, \mathsf{movi}, \mathsf{ld}, \mathsf{st}\}}{\Psi \vdash \{[\Delta].\,\Gamma\}\,\mathsf{c};\mathbb{I}}$$

By induction hypodissertation it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma'\urcorner\}\mathbb{I}$. By (9) it follows that $\ulcorner[\Delta].\,\Gamma\urcorner_{\Psi} \Rightarrow_{\mathsf{c}} \ulcorner[\Delta].\,\Gamma'\urcorner$. By rule SEQ it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathtt{c};\mathbb{I}$.

case JD.

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \vdash [\Delta].\,\Gamma \leq \Psi(\mathtt{f})}{\Psi \vdash \{[\Delta].\,\Gamma\}\,\mathsf{jd}\ \mathtt{f}}$$

By (8) it follows that $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \ulcorner\Psi(\mathtt{f})\urcorner$, and then $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \ulcorner\Psi\urcorner(\mathtt{f})$. By rule JD it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathsf{jd}\ \mathtt{f}$.

case JMP.

$$\frac{\Gamma(\mathtt{r}) = \mathsf{code}\ [\Delta'].\,\Gamma' \qquad \vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'}{\Psi \vdash \{[\Delta].\,\Gamma\}\,\mathsf{jmp}\ \mathtt{r}}$$

By translation it follows that $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \lambda(\mathbb{H},\mathbb{R}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\ulcorner[\Delta'].\,\Gamma'\urcorner)$. By (8) it follows that $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\,\Gamma'\urcorner$. By rule JMP it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathsf{jmp}\ \mathtt{r}$.

case MOVF.

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{[\Delta].\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{code}\ \ \Psi(\mathtt{f})\}\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\,\Gamma\}\,\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}}$$

By induction hypodissertation it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{code}\ \ \Psi(\mathtt{f})\}\urcorner\}\mathbb{I}$. By rule ECP and SEQ it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}$. ■

5.5 Discussion

We discussed the relationship between TAL and XCAP by showing a translation from TAL to XCAP. However, it is by no means limited to the translation itself.

For example, one can treat the translation as a shallow embeddeding of TAL types and typing rules in XCAP, which means there is no need to build meta theory for TAL while still letting the programmer work at the TAL level.

One interesting way to view the XCAP system is from TAL programmer's perspective. When writing programs that will interact with XCAP code that involve complex logical specification not expressible in TAL, so long as the interfaces abstract and hide that part of "real" logical specification (*e.g.*, by using existential packages), and local code behavior is simple enough (which is usually the case), the programmers can "pretend" that they are dealing with external TAL code instead of XCAP ones, by using the macros and lemmas defined for the translations. This lowers the requirement on programmers, as they do not need to learn XCAP, even if they are dealing with external XCAP code.

Chapter 6

A Port to x86 Machine

To demonstrate the potential of the XCAP framework, we want to use it to reason about realistic system code that actually runs the x86 machine, which is different from and more complex than the RISC-like target machine used in previous chapters. In this chapter, we present XCAP86, a port of the XCAP framework on Mini86, a faithful subset of the x86 architecture. XCAP86 and Mini86 adds the support of instruction decoding, finite machine word, word-aligned byte-addressed memory, conditional flags, built-in stack and push/pop instructions, and function call/return instructions.

The Mini86 machine is realistic, which brings additional complexities and proof engineering issues. Therefore, on top of XCAP86, we made practical adaptations and built useful abstractions, particularly on the handling of the stack and function calls. We demonstrate the usage of the XCAP86 and these abstractions for the verification of a polymorphic queue module, which is going to be used by the mini thread library in the next chapter.

6.1 Mini86: a Subset of the x86 Architecture

The execution environment of Mini86 (Figure 6.1) consists of a memory, a register file of eight general-purpose registers, a flags register made up of a carry bit and a zero bit, and a program counter. Following the x86 *Flat Model* [34], the memory of Mini86 appears

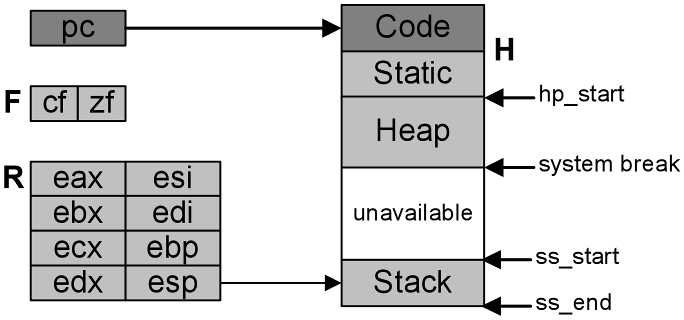


Figure 6.1: Mini86 execution environment

to a program as a single, continuous address space. Code, data (static and dynamic), and system stack all reside in this address space. Mini86 has a word size of 32 bits. Its memory is finite and ultimately restricted by the word size. The non-code part of the memory, the register file, and the flags register are referred to collectively as the machine state.

In comparison, the actual x86 execution environment uses an EFLAGS register to record a group of status and control flags including those mentioned above, and the eip (programmer counter) register can also be controlled implicitly by interrupts and exceptions, which we omit. We also omit the segment registers in x86, because they play no role in the Flat memory model.

The syntax of Mini86 is shown in Figure 6.2. In Mini86, two memory addressing modes are supported: a "direct" mode uses an immediate value as the absolute address; a "base indexed" mode uses a register value as the base address and an immediate value as the offset. Common instructions are included for arithmetic, data movement, comparison, control flow transfer, and stack manipulation.

An operand (o) is either an immediate value or a register. When an instruction takes two operands, the first one is the target operand. Conditional jumps are supported with help of conditional codes (a, b, and e stands for *above*, *below*, and *equal* respectively), which are represented by different combinations of cf and zf, and set by arithmetic and comparison instructions.

We present the operational semantics of Mini86 in Figure 6.3. All instructions are

$$\begin{array}{rrcl}
(\textit{Program}) & \mathbb{P} & ::= & (\mathbb{S}, pc) \\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{R}, \mathbb{F}) \\
(\textit{Mem}) & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^{*} \\
(\textit{Rfile}) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^{*} \\
(\textit{FReg}) & \mathbb{F} & ::= & \{\mathsf{cf} \rightsquigarrow \mathtt{b}, \mathsf{zf} \rightsquigarrow \mathtt{b}\} \\
(\textit{Word}) & \mathtt{w} & ::= & i \ \ (\textit{unsigned}\ 32\ \textit{bit integers}) \\
(\textit{Labels}) & \mathtt{l} & ::= & i \ \ (\textit{unsigned}\ 32\ \textit{bit integers},\ i\%4 = 0) \\
(\textit{CdLbl}) & \mathtt{f} & ::= & i \ \ (\textit{unsigned}\ 32\ \textit{bit integers}) \\
(\textit{Reg}) & \mathtt{r} & ::= & \mathsf{eax} \mid \mathsf{ebx} \mid \mathsf{ecx} \mid \mathsf{edx} \mid \mathsf{esi} \mid \mathsf{edi} \mid \mathsf{ebp} \mid \mathsf{esp} \\
(\textit{Bool}) & \mathtt{b} & ::= & tt \mid ff \\
(\textit{Cond}) & cc & ::= & \mathsf{a} \mid \mathsf{ae} \mid \mathsf{b} \mid \mathsf{be} \mid \mathsf{e} \mid \mathsf{ne} \\
(\textit{Addr}) & \mathtt{d} & ::= & i \mid \mathtt{r} \pm i \\
(\textit{Opr}) & \mathtt{o} & ::= & i \mid \mathtt{r} \\
(\textit{Instr}) & \mathtt{c} & ::= & \mathsf{add}\ \mathtt{r}, \mathtt{o} \mid \mathsf{sub}\ \mathtt{r}, \mathtt{o} \mid \mathsf{mov}\ \mathtt{r}, \mathtt{o} \mid \mathsf{mov}\ \mathtt{r}, [\mathtt{d}] \mid \mathsf{mov}\ [\mathtt{d}], \mathtt{o} \\
 & & & \mid \mathsf{cmp}\ \mathtt{r}, \mathtt{o} \mid \mathsf{j}cc\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{o} \mid \mathsf{push}\ \mathtt{o} \mid \mathsf{pop}\ \mathtt{r} \mid \mathsf{pop} \mid \mathsf{call}\ \mathtt{o} \mid \mathsf{ret}
\end{array}$$

Figure 6.2: Mini86 syntax

encoded as machine words and stored in the memory; at each step, the machine has to decode the instruction from the memory at *pc*. During execution, the machine would fetch words based on the program counter *pc* and decode the words before executing it.

We use Dc to fetch and decode an instruction out of the memory $\mathbb{H}$ given a program counter *pc*. The result of Dc is an instruction c and a new program counter *npc*. We also use a macro $\mathtt{Next}_{\mathsf{c}}(\mathbb{S})$ to define the transition of the machine state on some of the instructions. It is worth noting that this macro provides only a partial mapping: there are cases where no valid next states are defined. Examples include accessing invalid memory or stack locations. The Mini86 machine gets stuck in these cases.

The stack instructions of Mini86 assume that the stack pointer is held by esp. As a general rule, esp should not be used for any other purposes. For example, the push o instruction decrements esp by 4 and writes the value of o onto the new top of the stack. The pop r instruction reads a value from the top of the stack, puts it into r, and increments esp by 4.

Suppose $\mathsf{Dc}(\mathbb{H},pc)=(\mathtt{c},npc)$	
if $\mathtt{c}=$	then $((\mathbb{H},\mathbb{R},\mathbb{F}),pc)\longmapsto$
jmp $\mathtt{o}$	$(\mathbb{H},\mathbb{R})\hat{\mathbb{R}}(\mathtt{o})$
jcc $\mathtt{f}$	if $\hat{\mathbb{F}}(cc)$ then $(\mathbb{H},\mathbb{R})\mathtt{f}$ else $(\mathbb{H},\mathbb{R})npc$ where suppose $\mathbb{F}=\{\mathsf{cf}\rightsquigarrow cf,\mathsf{zf}\rightsquigarrow zf\}$, we have $\hat{\mathbb{F}}(\mathsf{a})\triangleq\neg cf\wedge\neg zf,\ \hat{\mathbb{F}}(\mathsf{ae})\triangleq\neg cf,\ \hat{\mathbb{F}}(\mathsf{b})\triangleq cf,$ $\hat{\mathbb{F}}(\mathsf{be})\triangleq cf\vee zf,\quad \hat{\mathbb{F}}(\mathsf{e})\triangleq zf,\quad \hat{\mathbb{F}}(\mathsf{ne})\triangleq\neg zf$
call $\mathtt{o}$	$(\mathtt{Next}_{\mathsf{push}\ npc}(\mathbb{H},\mathbb{R},\mathbb{F}),\hat{\mathbb{R}}(\mathtt{o}))$
ret	$(\mathtt{Next}_{\mathsf{pop}}(\mathbb{H},\mathbb{R},\mathbb{F}),\mathbb{H}(sp))$ when $sp\in\mathsf{dom}(\mathbb{H})$
$\mathtt{c}$	$(\mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R},\mathbb{F}),npc)$

where

if $\mathtt{c}=$	then $\mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R},\mathbb{F})=$
add $\mathtt{r},\mathtt{o}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow\mathbb{R}(\mathtt{r})+\hat{\mathbb{R}}(\mathtt{o})\},\mathsf{CalcF}(\mathbb{R}(\mathtt{r})+\hat{\mathbb{R}}(\mathtt{o})))$ when $0\leq\mathbb{R}(\mathtt{r})+\hat{\mathbb{R}}(\mathtt{o})<2^{32}$
sub $\mathtt{r},\mathtt{o}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow\mathbb{R}(\mathtt{r})-\hat{\mathbb{R}}(\mathtt{o})\},\mathsf{CalcF}(\mathbb{R}(\mathtt{r})-\hat{\mathbb{R}}(\mathtt{o})))$ when $0\leq\mathbb{R}(\mathtt{r})-\hat{\mathbb{R}}(\mathtt{o})<2^{32}$
cmp $\mathtt{r},\mathtt{o}$	$(\mathbb{H},\mathbb{R},\mathsf{CalcF}(\mathbb{R}(\mathtt{r})-\hat{\mathbb{R}}(\mathtt{o})))$
mov $\mathtt{r},\mathtt{o}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow\hat{\mathbb{R}}(\mathtt{o})\},\mathbb{F})$
mov $\mathtt{r},[\mathtt{d}]$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow\mathbb{H}(\hat{\mathbb{R}}(\mathtt{d}))\},\mathbb{F})$ when $\hat{\mathbb{R}}(\mathtt{d})\in\mathsf{dom}(\mathbb{H})$
mov $[\mathtt{d}],\mathtt{o}$	$(\mathbb{H}\{\hat{\mathbb{R}}(\mathtt{d})\rightsquigarrow\hat{\mathbb{R}}(\mathtt{o})\},\mathbb{R},\mathbb{F})$ when $\hat{\mathbb{R}}(\mathtt{d})\in\mathsf{dom}(\mathbb{H})$
push $\mathtt{o}$	$(\mathbb{H}\{sp-4\rightsquigarrow\hat{\mathbb{R}}(\mathtt{o})\},\ \mathbb{R}\{\mathsf{esp}\rightsquigarrow sp-4\},\mathbb{F})$ when $sp-4\in\mathsf{dom}(\mathbb{H})$
pop $\mathtt{r}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow\mathbb{H}(sp)\}\{\mathsf{esp}\rightsquigarrow sp+4\},\ \mathbb{F})$ when $sp\in\mathsf{dom}(\mathbb{H})$ and $0\leq sp+4<2^{32}$
pop	$(\mathbb{H},\mathbb{R}\{\mathsf{esp}\rightsquigarrow sp+4\},\ \mathbb{F})$ when $0\leq sp+4<2^{32}$

$\mathsf{Dc}()$ is the instruction decoding function

$$\hat{\mathbb{R}}(i)\triangleq i\qquad \hat{\mathbb{R}}(\mathtt{r})\triangleq\mathbb{R}(\mathtt{r})\qquad \hat{\mathbb{R}}(\mathtt{r}\pm i)\triangleq\mathbb{R}(\mathtt{r})\pm i$$

$$\mathsf{CalcF}(i)\triangleq\{\mathsf{cf}\mapsto i<0,\mathsf{zf}\mapsto i=0\}$$

Figure 6.3: Dynamic semantics of Mini86

The call instruction pushes the return address (calculated from *pc*) onto the stack and transfers the control to the callee (by updating *pc*). The ret instruction bump the stack pointer by 4 and transfer the control to the return address. Note that the ret instruction does not necessarily transfer the control back to the caller, as the program may modify the stack in arbitrary ways. Such maneuver is indeed commonly used in implementing thread primitives.

6.2 XCAP86: a Port of XCAP on Mini86

In this section we first introduce the XCAP86 language, where most of the constructs and rules are the same as XCAP. We then discuss how to abstract and reason about memory, stack, function call/return interfaces.

Assertion language. The syntax of XCAP86 is given in Figure 6.4. We abstract a code heap $\mathbb{C}$ out of the actual memory $\mathbb{H}$ of Mini86. Although the code heap is presented in the syntax as a mapping from code labels to instruction sequences, it is actually embedded in the memory by encoding instructions as machine words. XCAP86 includes features such as embedded code pointers, impredicative polymorphisms, and recursive specifications.

We present the validity rules of XCAP86 extended propositions in Figure 6.5. Following XCAP, the interpretation of extended propositions is defined as their validity under the empty environment:

$$[\![\mathtt{P}]\!]_{\Psi} \triangleq \cdot \vdash_{\Psi} \mathtt{P}$$

Following XCAP, we establish the following soundness theorem of the interpretation of extended propositions (with respect to CiC/Coq).

Theorem 6.1 (Soundness of XCAP86 *PropX* Interpretation)

1. If $[\![\, \langle p \rangle \,]\!]_{\Psi}$ then p;

2. if $[\![\mathsf{cptr}(\mathtt{f},\mathtt{a})]\!]_{\Psi}$ then $\Psi(\mathtt{f}) = \mathtt{a}$;

$$
\begin{array}{rcll}
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \leadsto \mathbb{I}\}^{*} & \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \mathtt{c} \mid \mathtt{c};[\mathtt{f}] \mid \mathtt{c};\mathbb{I} & \\
(\mathit{PropX}) & \mathtt{P},\mathtt{Q} & ::= \langle p \rangle & \mathit{lifted\ meta\ proposition} \\
 & & \quad \mid \mathsf{cptr}(\mathtt{f},\mathtt{a}) & \mathit{embedded\ code\ pointer} \\
 & & \quad \mid \mathtt{P} \text{⩓} \mathtt{Q} & \mathit{conjunction} \\
 & & \quad \mid \mathtt{P} \text{⩔} \mathtt{Q} & \mathit{disjunction} \\
 & & \quad \mid \mathtt{P} \twoheadrightarrow \mathtt{Q} & \mathit{implication} \\
 & & \quad \mid \text{∀̸}\!\!\forall x{:}A.\,\mathtt{P} & \mathit{universal\ quantification} \\
 & & \quad \mid \exists\!\!\exists x{:}A.\,\mathtt{P} & \mathit{existential\ quantification} \\
 & & \quad \mid \forall\!\!\forall \mathtt{a}{:}A \to \mathit{PropX}.\,\mathtt{P} & \mathit{imp.\ universal\ quan.} \\
 & & \quad \mid \exists\!\!\exists \mathtt{a}{:}A \to \mathit{PropX}.\,\mathtt{P} & \mathit{imp.\ existential\ quan.} \\
 & & \quad \mid (\boldsymbol{\mu}\,\alpha.\,\lambda x{:}A.\,\mathtt{P}\ \ B) & \mathit{recursive\ definition} \\
(\mathit{CdHpSpec}) & \Psi & ::= \{\mathtt{f} \leadsto \mathtt{a}\}^{*} & \\
(\mathit{Assertion}) & \mathtt{a} & \in \mathit{State} \to \mathit{PropX} & \\
(\mathit{AssertImp}) & \mathtt{a} \Rightarrow \mathtt{a}' & \triangleq \forall \Psi, \mathbb{S}.\, \llbracket \mathtt{a} \rrbracket_{\Psi}\, \mathbb{S} \supset \llbracket \mathtt{a}' \rrbracket_{\Psi}\, \mathbb{S} & \\
(\mathit{StepImp}) & \mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' & \triangleq \forall \Psi, \mathbb{S}.\, \llbracket \mathtt{a} \rrbracket_{\Psi}\, \mathbb{S} \supset \llbracket \mathtt{a}' \rrbracket_{\Psi}\, \mathtt{Next}_{\mathtt{c}}(\mathbb{S}) &
\end{array}
$$

Figure 6.4: Syntax of XCAP86

$\boxed{\Gamma \vdash_\Psi \texttt{P}}$ **(*Validity of Extended Propositions*)** $\qquad$ *(env)* $\Gamma := \cdot \mid \Gamma, \texttt{P}$

(The following presentation omits the Ψ in judgment $\Gamma \vdash_\Psi P$.)

$$\frac{\texttt{P} \in \Gamma}{\Gamma \vdash \texttt{P}}\ (\textsc{env}) \qquad \frac{p}{\Gamma \vdash \langle p \rangle}\ (\langle\rangle\text{-I}) \qquad \frac{\Gamma \vdash \langle p \rangle \qquad p \supset (\Gamma \vdash \texttt{Q})}{\Gamma \vdash \texttt{Q}}\ (\langle\rangle\text{-E})$$

$$\frac{\Psi(\texttt{f}) = \texttt{a}}{\Gamma \vdash \mathsf{cptr}(\texttt{f}, \texttt{a})}\ (\textsc{cp-i}) \qquad \frac{\Gamma \vdash \mathsf{cptr}(\texttt{f}, \texttt{a}) \qquad (\Psi(\texttt{f}) = \texttt{a}) \supset (\Gamma \vdash \texttt{Q})}{\Gamma \vdash \texttt{Q}}\ (\textsc{cp-e})$$

$$\frac{\Gamma \vdash \texttt{P} \qquad \Gamma \vdash \texttt{Q}}{\Gamma \vdash \texttt{P} \Wedge \texttt{Q}}\ (\Wedge\text{-I}) \qquad \frac{\Gamma \vdash \texttt{P} \Wedge \texttt{Q}}{\Gamma \vdash \texttt{P}}\ (\Wedge\text{-E1}) \qquad \frac{\Gamma \vdash \texttt{P} \Wedge \texttt{Q}}{\Gamma \vdash \texttt{Q}}\ (\Wedge\text{-E2})$$

$$\frac{\Gamma \vdash \texttt{P}}{\Gamma \vdash \texttt{P} \Vee \texttt{Q}}\ (\Vee\text{-I1}) \qquad \frac{\Gamma \vdash \texttt{Q}}{\Gamma \vdash \texttt{P} \Vee \texttt{Q}}\ (\Vee\text{-I2}) \qquad \frac{\Gamma \vdash \texttt{P} \Vee \texttt{Q} \qquad \Gamma, \texttt{P} \vdash \texttt{R} \qquad \Gamma, \texttt{Q} \vdash \texttt{R}}{\Gamma \vdash \texttt{R}}\ (\Vee\text{-E})$$

$$\frac{\Gamma, \texttt{P} \vdash \texttt{Q}}{\Gamma \vdash \texttt{P} \twoheadrightarrow \texttt{Q}}\ (\twoheadrightarrow\text{-I}) \qquad \frac{\Gamma \vdash \texttt{P} \twoheadrightarrow \texttt{Q} \qquad \Gamma \vdash \texttt{P}}{\Gamma \vdash \texttt{Q}}\ (\twoheadrightarrow\text{-E})$$

$$\frac{\Gamma \vdash \texttt{P}[B/x] \qquad \forall\, B : A}{\Gamma \vdash \forall\!\!\!\forall x : A.\texttt{P}}\ (\forall\!\!\!\forall\text{-I1}) \qquad \frac{\Gamma \vdash \forall\!\!\!\forall x : A.\texttt{P} \qquad B : A}{\Gamma \vdash \texttt{P}[B/x]}\ (\forall\!\!\!\forall\text{-E1})$$

$$\frac{B : A \qquad \Gamma \vdash \texttt{P}[B/x]}{\Gamma \vdash \exists\!\!\!\exists x : A.\texttt{P}}\ (\exists\!\!\!\exists\text{-I1}) \qquad \frac{\Gamma \vdash \exists\!\!\!\exists x : A.\texttt{P} \qquad \Gamma, \texttt{P}[B/x] \vdash \texttt{Q} \qquad \forall\, B : A}{\Gamma \vdash \texttt{Q}}\ (\exists\!\!\!\exists\text{-E1})$$

$$\frac{\Gamma \vdash \texttt{P}[\texttt{a}/\alpha] \qquad \forall\, \texttt{a} : A \to \mathit{PropX}}{\Gamma \vdash \forall\!\!\!\forall \alpha : A \to \mathit{PropX}.\texttt{P}}\ (\forall\!\!\!\forall\text{-I2}) \qquad \frac{\texttt{a} : A \to \mathit{PropX} \qquad \Gamma \vdash \texttt{P}[\texttt{a}/\alpha]}{\Gamma \vdash \exists\!\!\!\exists \alpha : A \to \mathit{PropX}.\texttt{P}}\ (\exists\!\!\!\exists\text{-I2})$$

$$\frac{B : A \qquad \Gamma \vdash \texttt{P}[B/x][\boldsymbol{\mu}\alpha : A \to \mathit{PropX}.\lambda x : A.\texttt{P}/\alpha]}{\Gamma \vdash (\boldsymbol{\mu}\, \alpha : A \to \mathit{PropX}.\lambda x : A.\texttt{P}\ \ B)}\ (\mu\text{-I})$$

Figure 6.5: Validity rules of XCAP86 extended propositions

3. if $[\![\mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q}]\!]_\Psi$ then $[\![\mathtt{P}]\!]_\Psi$ and $[\![\mathtt{Q}]\!]_\Psi$;

4. if $[\![\mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q}]\!]_\Psi$ then either $[\![\mathtt{P}]\!]_\Psi$ or $[\![\mathtt{Q}]\!]_\Psi$;

5. if $[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_\Psi$ and $[\![\mathtt{P}]\!]_\Psi$ then $[\![\mathtt{Q}]\!]_\Psi$;

6. if $[\![\mathbb{\forall} x{:}A.\,\mathtt{P}]\!]_\Psi$ and $B{:}A$ then $[\![\mathtt{P}[B/x]\,]\!]_\Psi$;

7. if $[\![\,\mathbb{\exists} x{:}A.\,\mathtt{P}]\!]_\Psi$ then there exists $B{:}A$ such that $[\![\mathtt{P}[B/x]\,]\!]_\Psi$;

8. if $[\![\mathbb{\forall} \alpha{:}A \to PropX.\,\mathtt{P}]\!]_\Psi$ and $\mathtt{a}{:}A \to PropX$ then $[\![\mathtt{P}[\mathtt{a}/\alpha]\,]\!]_\Psi$;

9. if $[\![\,\mathbb{\exists} \alpha{:}A \to PropX.\,\mathtt{P}]\!]_\Psi$ then there exists $\mathtt{a}{:}A \to PropX$ such that $[\![\mathtt{P}[\mathtt{a}/\alpha]\,]\!]_\Psi$;

10. if $[\![\,(\boldsymbol{\mu}\,\alpha.\,\lambda x{:}A.\,\mathtt{P}\;\;B)\,]\!]_\Psi$ then $[\![\mathtt{P}[B/x][(\boldsymbol{\mu}\,\alpha.\,\lambda x{:}A.\,\mathtt{P}\;\;)/\alpha]\,]\!]_\Psi$.

Corollary 6.2 (XCAP86 Consistency) $[\![\,\langle\mathsf{False}\rangle\,]\!]_\Psi$ is not provable.

Inference rules. The major difference between XCAP86 and XCAP is on the inference rules, which is presented in Figure 6.6. In the top-level well-formed program rule, $\mathsf{DC}(\mathbb{C})$ is a predicate that establishes the proper instruction decoding relation between the code heap $\mathbb{C}$ and the memory $\mathbb{H}$. Its implementation makes use of the single-instruction decoding function $\mathsf{Dc}()$ that appears in the previous section. $\mathsf{lookup}(\mathbb{C},\mathtt{f},\mathbb{I})$ is a macro that checks whether the instruction sequence $\mathbb{I}$ is inside code heap $\mathbb{C}$ at location $\mathtt{f}$.

Using cptr, XCAP86 supports the reasoning of embedded code pointers (ECP) in general. The idea can be naturally adapted according to the x86 additional instructions that use ECPs. In particular, there are now three new rules for the function call and return instructions of Mini86, assuming a built-in stack. A call instruction pushes a return address onto the stack and transfers the control to the target code. In our actual implementation, the return address is calculated from the *pc*. To avoid obfuscating the presentation, we used an explicit $[\mathtt{f}_{ret}]$ in the above Rule CALLI. This rule says, if $\mathtt{a}$ holds on the current state, then $\Psi(\mathtt{f})$ holds on the updated state after executing the stack push. The Rule RET

$\boxed{\Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{P}}$ ***(Well-formed Program)***

$$\frac{\Psi_{G} \vdash \mathbb{C} : \Psi_{G} \qquad ((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_{G}})\ \mathbb{S}) \qquad \mathsf{lookup}(\mathbb{C}, pc, \mathbb{I}) \qquad \Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi_{G} \vdash \{\mathtt{a}\}\,(\mathbb{S}, pc)} \; (\textsc{prog})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C} : \Psi}$ ***(Well-formed Code Heap)***

$$\frac{\Psi_{IN} \vdash \{\mathtt{a}_i\}\,\mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \rightsquigarrow \mathbb{I}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathbb{I}_n\} : \{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\}} \; (\textsc{cdhp})$$

$$\frac{\begin{array}{c}\Psi_{IN\,1} \vdash \mathbb{C}_1 : \Psi_1 \qquad \Psi_{IN\,2} \vdash \mathbb{C}_2 : \Psi_2 \qquad \Psi_{IN\,1}(\mathtt{f}) = \Psi_{IN\,2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN\,1}) \cap \mathsf{dom}(\Psi_{IN\,2})\end{array}}{\Psi_{IN\,1} \cup \Psi_{IN\,2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2} \; (\textsc{link})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}$ ***(Well-formed Instruction Sequence)***

$$\frac{\mathtt{a} \Rightarrow_{\mathsf{c}} \mathtt{a}' \qquad \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add}, \mathsf{sub}, \mathsf{cmp}, \mathsf{mov}, \mathsf{push}, \mathsf{pop}\}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{c};\mathbb{I}} \; (\textsc{seq})$$

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{f}} \; (\textsc{jmpi})$$

$$\frac{\begin{array}{c}(\lambda(\mathbb{H}, \mathbb{R}, \mathbb{F}).\, \langle \neg \hat{\mathbb{F}}(cc) \rangle \ \wedge\!\!\!\wedge\ \mathtt{a}\ (\mathbb{H}, \mathbb{R}, \mathbb{F})) \Rightarrow \mathtt{a}' \qquad \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I} \\ (\lambda(\mathbb{H}, \mathbb{R}, \mathbb{F}).\, \langle \hat{\mathbb{F}}(cc) \rangle \ \wedge\!\!\!\wedge\ \mathtt{a}\ (\mathbb{H}, \mathbb{R}, \mathbb{F})) \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{j}cc\ \mathtt{f};\mathbb{I}} \; (\textsc{jcc})$$

$$\frac{\mathtt{a} \Rightarrow \lambda(\mathbb{H}, \mathbb{R}, \mathbb{F}).\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}') \qquad \mathtt{a} \Rightarrow \mathtt{a}'}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}} \; (\textsc{jmpr})$$

$$\frac{(\lambda \mathbb{S}.\, \mathsf{cptr}(\mathtt{f}, \Psi(\mathtt{f})) \wedge\!\!\!\wedge\, \mathtt{a}\ \mathbb{S}) \Rightarrow \mathtt{a}' \qquad \mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I}}{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}} \; (\textsc{ecp})$$

$$\frac{\mathtt{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{call}\ \mathtt{f}; [\mathtt{f}_{ret}]} \; (\textsc{calli})$$

$$\frac{\mathtt{a} \Rightarrow \lambda(\mathbb{H}, \mathbb{R}, \mathbb{F}).\, \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}') \qquad \mathtt{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \mathtt{a}'}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{call}\ \mathtt{r}; [\mathtt{f}_{ret}]} \; (\textsc{callr})$$

$$\frac{\mathtt{a} \Rightarrow \lambda(\mathbb{H}, \mathbb{R}, \mathbb{F}).\, \mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})), \mathtt{a}') \qquad \mathtt{a} \Rightarrow_{\mathsf{pop}} \mathtt{a}'}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{ret}} \; (\textsc{ret})$$

Figure 6.6: Inference rules of XCAP86

says, the top of the stack is a code pointer with pre-condition $\mathtt{a}'$ and, if $\mathtt{a}$ holds on the current state, $\mathtt{a}'$ holds on the updated state after popping out the return address.

It is worth noting that Rule CALLI does not enforce the validity of the return address. This allows some "fake" function calls that never return, a pattern indeed used in the thread library in the next chapter. Specialized derived rules can be built as lemmas to reflect regular function calls.

Soundness. Following XCAP, we establish the soundness of these inferences rules with respect to the Mini86 operational semantics.

Lemma 6.3 (XCAP86 Progress)
If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Lemma 6.4 (XCAP86 Preservation)
If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$ then there exists an assertion $\mathtt{a}'$ such that $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{P}'$.

Theorem 6.5 (XCAP86 Soundness)
If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P}\longmapsto^n \mathbb{P}'$.

For the presentation of the rest of this chapter and the next chapter, we ignore the difference between *Prop* and *PropX*, and always use the syntax of *Prop* terms for *PropX* terms and μ for $\boldsymbol{\mu}$.

Reasoning about memory. The reasoning on memory (data heap and stack) operations is largely carried out following separation logic [48, 53]. In particular, the primitives of separation logic are defined as shorthands using the primitives of the underlying logic in XCAP (a shallow embedding of separation logic in the assertion language). Some representative cases are given as follows.

$$\begin{aligned}
\mathsf{emp} &\triangleq \lambda\mathbb{H}.\,\mathbb{H}=\{\}\\
\mathtt{l}\mapsto\mathtt{w} &\triangleq \lambda\mathbb{H}.\,\mathtt{l}\neq\mathsf{NULL}\wedge\mathbb{H}=\{\mathtt{l}\leadsto\mathtt{w}\}\\
\mathtt{l}\mapsto\_ &\triangleq \lambda\mathbb{H}.\,\exists\mathtt{w}.\,(\mathtt{l}\mapsto\mathtt{w}\;\mathbb{H})\\
\mathtt{a}_1*\mathtt{a}_2 &\triangleq \lambda\mathbb{H}.\,\exists\mathbb{H}_1,\mathbb{H}_2.\,\mathbb{H}_1\uplus\mathbb{H}_2=\mathbb{H}\wedge\mathtt{a}_1\;\mathbb{H}_1\wedge\mathtt{a}_2\;\mathbb{H}_2\\
\mathtt{l}\mapsto\mathtt{w}_1,\ldots,\mathtt{w}_n &\triangleq \mathtt{l}\mapsto\mathtt{w}_1\;*\;\mathtt{l}+4\mapsto\mathtt{w}_2\;*\;\ldots\;*\;\mathtt{l}+4(n-1)\mapsto\mathtt{w}_n\\
\mathtt{l}\mapsto[n] &\triangleq \mathtt{l}\mapsto\_,\ldots,\_\qquad\textit{(the number of \_ is } n/4)
\end{aligned}$$

emp asserts that the memory is empty. $\mathtt{l}\mapsto\mathtt{w}$ asserts that the memory contains exactly one word at address $\mathtt{l}$ with the value $\mathtt{w}$; when the value is not important, a wildcard is used as in $\mathtt{l}\mapsto\_$. Separating conjunction $\mathtt{a}*\mathtt{a}'$ asserts that the memory can be split into two disjoint parts in which $\mathtt{a}$ and $\mathtt{a}'$ hold respectively ($\uplus$ represents disjoint union). $\mathtt{l}\mapsto\mathtt{w}_1,\ldots,\mathtt{w}_n$ asserts that $\mathtt{l}$ is the starting address of a sequence of words $\mathtt{w}_1,\ldots,\mathtt{w}_n$; our memory is addressed by bytes, hence the number 4 (one word is 4 bytes). Finally, $\mathtt{l}\mapsto[n]$ means that $\mathtt{l}$ points to n bytes. For conciseness, we sometimes omit lambda and existential bindings of variables, thus writing $\lambda x_1,\ldots,x_n.\exists y_1,\ldots,y_m.\mathtt{P}$ simply as $\mathtt{P}$ when there is no confusion. We also sometimes simplify $(\mathtt{a}*\mathtt{a}'\;\mathbb{H})$ as $\mathtt{a}*\mathtt{a}'$.

Stack and calling convention. Besides specializing XCAP for our machine model, we also built key abstractions to help manage the complexity of the reasoning, such as the handling of the stack and calling convention.

Most Mini86 code follows the convention illustrated in Figure 6.7. The return address is on stack top [esp]. Function arguments follow in [esp+4], [esp+8], *etc.*. The return value is stored in eax. Unless specified otherwise, ebx, esi, edi and ebp are callee-save registers.

It is therefore convenient to built a specification template reflecting this convention. For a function with n arguments $a_1\ldots a_n$, we write its specification (*i.e.,* a pre-condition in the form of an assertion) as:

$$\mathsf{Fn}\;\; a_1,\ldots,a_n\;\{\mathsf{Aux}:x_1,\ldots,x_m;\;\mathsf{Local}:[fs];\;\mathsf{Pre}:\mathsf{a}_{pre};\;\mathsf{Post}:\mathsf{a}_{post}\}$$

The intention of this macro is that $x_1,\ldots,x_m$ are "auxiliary variables" commonly used in

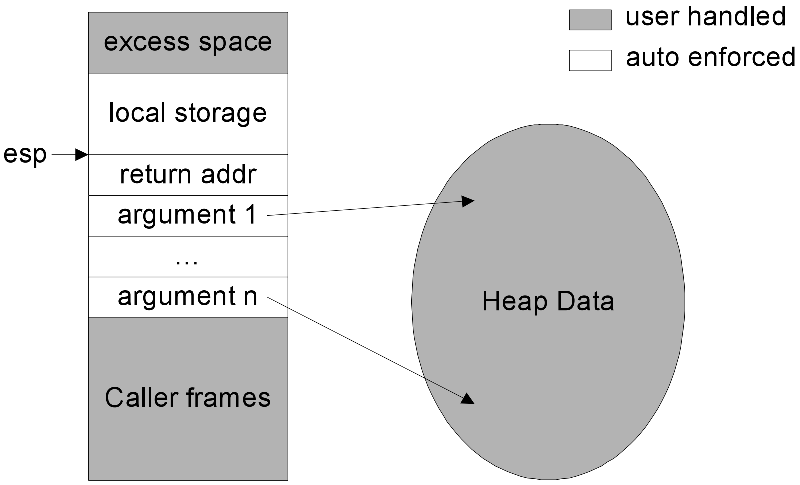


Figure 6.7: Function calling convention

Hoare-logic style reasoning, fs is the size of required free space on the stack, and a_{pre} and a_{post} are the pre- and post-conditions of the function. We sometimes refer to the variables collectively as $\vec{a}$ and $\vec{x}$. This macro is defined as follows:

$$\begin{aligned}
&\exists a_1,\ldots,a_n, cs, sp, ss, ret, x_1,\ldots,x_m, \mathsf{a}_{prv}.\\
&\qquad\qquad\qquad \mathsf{reg}(cs,sp) \ \land\ ss \geq fs\\
&\qquad\qquad \land\ \mathsf{stack}(sp,ss,ret,a_1,\ldots,a_n) \quad * \ \mathsf{a}_{prv} \ * \ \mathsf{a}_{pre}\\
&\ \land\ \mathsf{cptr}(ret,\ \exists retv.\ \mathsf{reg}(cs,sp{+}4) \land \mathsf{eax}{=}retv\\
&\qquad\qquad \land\ \mathsf{stack}(sp{+}4,ss{+}4,a_1,\ldots,a_n) \ * \ \mathsf{a}_{prv} \ * \ \mathsf{a}_{post})
\end{aligned}$$

The first line of this definition quantifies over the values of (1) function arguments $a_1,\ldots,a_n$, (2) callee-save registers cs (a 4-tuple), (3) the stack pointer sp, (4) the size of available space on stack ss, (5) the return address ret, (6) auxiliary variables $x_1,\ldots,x_m$, and (7) some hidden private data expressed as the predicate a_{prv}.

The second line relates the register file with the callee-save values cs (4-tuple) and the stack pointer sp using the macro below. It also makes sure that there is enough space available on the stack.

$$\mathsf{reg}(ebx,esi,edi,ebp,esp) \triangleq \mathsf{ebx}{=}ebx \land \mathsf{esi}{=}esi \land \mathsf{edi}{=}edi \land \mathsf{ebp}{=}ebp \land \mathsf{esp}{=}esp$$

The third line describes (1) the stack frame upon entering the function using the macro below, (2) the private data hidden from the function, and (3) the user customized precondition a_{pre}, which does not directly talk about register files and the current stack frame, since they have already been handled by the calling convention.

$$\mathsf{stack}(sp, ss, w_1, \ldots, w_n) \triangleq sp{-}ss \mapsto [ss] \, * \, sp \mapsto w_1, \ldots., w_n.$$

The last two lines of the Fn definition specify the return address *ret* as an embedded code pointer using cptr. When a function returns, the callee-save registers, stack frame, and private data must all be preserved, and the post-condition a_{post} must be established. Note that (1) eax may contain a return value *retv*, and (2) the return instruction automatically increases the stack pointer by 4.

We built another abstractions to help manage the complexity of verification. It is a finer-grained version of Fn which, besides following the convention above, also specifies local variables on the stack by extending the "$\mathsf{Local}: [fs]$;" part to "$\mathsf{Local}: [fs], v_1, \ldots, v_k$;", where v_i are the local values on the stack. The definition of this new Fn is:

$$\begin{array}{l}
\exists a_1, \ldots, a_n, v_1, \ldots, v_k, cs, sp, ss, ret, x_1, \ldots, x_m, \mathsf{a}_{prv}. \\
\qquad\qquad\qquad \mathsf{reg}(cs, sp) \wedge ss \geq fs \\
\qquad\quad \wedge\ \mathsf{stack}(sp, ss, v_1, \ldots, v_k, ret, a_1, \ldots, a_n) \ \ * \ \mathsf{a}_{prv} \ * \ \mathsf{a}_{pre} \\
\quad \wedge\ \mathsf{cptr}(ret,\ \exists retv.\ \mathsf{reg}(cs, sp{+}4) \wedge \mathsf{eax} = retv \\
\qquad\quad \wedge\ \mathsf{stack}(sp{+}4{+}4k, ss{+}4{+}4k, a_1, \ldots, a_n) \ * \ \mathsf{a}_{prv} \ * \ \mathsf{a}_{post})
\end{array}$$

We define many derived rules (implemented as lemmas) to facilitate the reasoning related to functions specified using the new Fn. We provide the following examples:

$$\frac{\mathsf{a}_{pre} \Rightarrow_{\mathsf{C}} \mathsf{a} \qquad \Psi \vdash \{\mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}: \vec{x}; \mathsf{Local}: [fs], \vec{v}; \mathsf{Pre}: \mathsf{a}; \mathsf{Post}: \mathsf{a}_{post}\}\}\, \mathbb{I}}{\Psi \vdash \{\mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}: \vec{x}; \mathsf{Local}: [fs], \vec{v}; \mathsf{Pre}: \mathsf{a}_{pre}; \mathsf{Post}: \mathsf{a}_{post}\}\}\ \mathtt{c}; \mathbb{I}}\ (\textsc{fn-seq})$$

$$\frac{fs \geq 4 \qquad \Psi \vdash \{\mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}: \vec{x}; \mathsf{Local}: [fs{-}4], \mathtt{w}, \vec{v}; \mathsf{Pre}: \mathsf{a}_{pre}; \mathsf{Post}: \mathsf{a}_{post}\}\}\, \mathbb{I}}{\Psi \vdash \{\mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}: \vec{x}; \mathsf{Local}: [fs], \vec{v}; \mathsf{Pre}: \mathsf{a}_{pre}; \mathsf{Post}: \mathsf{a}_{post}\}\}\ \mathsf{push}\ \mathtt{w}; \mathbb{I}}\ (\textsc{fn-push})$$

$$\frac{\begin{array}{l}\Psi(\mathtt{f}) = \mathsf{Fn}\ \vec{a}'\ \{\mathsf{Aux}:\vec{x}';\mathsf{Local}:[fs'];\mathsf{Pre}:\mathsf{a}_{pre}';\mathsf{Post}:\mathsf{a}_{post}'\}\\ \Psi(\mathtt{f}_{ret}) = \mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}:\vec{x};\mathsf{Local}:[fs],\vec{v};\mathsf{Pre}:\mathsf{a};\mathsf{Post}:\mathsf{a}_{post}\}\\ \mathsf{a}_{pre} \Rightarrow (\mathsf{a}_{prv} * \mathsf{a}_{pre}' \wedge (\mathsf{a}_{prv} * \mathsf{a}_{post}' \Rightarrow \mathsf{a}))\\ fs-4 > fs' \qquad \mathtt{f} \in \mathsf{dom}(\Psi) \qquad \mathtt{f}_{ret} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}:\vec{x};\mathsf{Local}:[fs],\vec{v};\mathsf{Pre}:\mathsf{a}_{pre};\mathsf{Post}:\mathsf{a}_{post}\}\}\ \mathsf{call}\ \mathtt{f};[\mathtt{f}_{ret}]}\ (\text{FN-CALLI})$$

$$\frac{\mathsf{a}_{pre} \Rightarrow \mathsf{a}_{post}}{\Psi \vdash \{\mathsf{Fn}\ \vec{a}\ \{\mathsf{Aux}:\vec{x};\mathsf{Local}:[fs];\mathsf{Pre}:\mathsf{a}_{pre};\mathsf{Post}:\mathsf{a}_{post}\}\}\ \mathsf{ret}}\ (\text{FN-RET})$$

6.3 Verification of a polymorphic queue module

We demonstrate some of those macros defined in the previous section with the specification and verification of a polymorphic queue module, which will be used by the thread library in the next chapter. Our queue module's C specification as follows.

```
typedef struct node_st *node_t;
struct node_st node_t next;

void queue_insert (node_t *q, node_t t);
node_t queue_delete (node_t *q);
```

The modules defines a node data structure and functions for queue insertion and remove.

We define the XCAP86 data type for the queue noder pointer as follows:

$$\begin{aligned}\mathsf{node\_t\_ref}(\mathsf{a},\mathsf{nil},q) &\triangleq q \mapsto \mathsf{NULL}\\ \mathsf{node\_t\_ref}(\mathsf{a},t::tl,q) &\triangleq q \mapsto t \ * \ \mathsf{a}(t) \ * \ \mathsf{node\_t\_ref}(\mathsf{a},tl,t)\end{aligned}$$

This defines a data structure of polymorphic queues, where the link field of a queue node is always stored in the first word of the node. More specifically, $\mathsf{node\_t\_ref}(\mathsf{a},[t],q)$ represents that q is a pointer to a queue; the locations of the queue nodes are collected as a list $[t]$, and every queue node satisfies the predicate a. The definition is inductive. In the base case, an empty queue specifies just the queue pointer q itself, which points to NULL. In the inductive case, a queue comprises of a head node at t satisfying a and a tail queue of $\mathsf{node\_t\_ref}(\mathsf{a},tl,t)$.

queue_insert will insert a properly shaped node into the end of the queue. *queue_delete*() returns NULL when the queue is empty, otherwise, it returns the first node of the queue and removes it from the queue. We present the code and verification of *queue_insert*() and *queue_delete*() in Figure 6.8 and Figure 6.9. In the figures, there are assembly code, comments, function interfaces, and assertions at each key program point.

As the next chapter will show, the polymorphic queue module can be used by multiple modules, which demonstrates the modularity of its specifications.

Fn q,t { Aux: a,tl; Local: $[0]$
Pre: node_t_ref$(\mathsf{a},tl,q) * t \mapsto _ * \mathsf{a}\ t$;
Post: node_t_ref$(\mathsf{a},tl@[t],q)$ }

```
queue_insert:                        // void queue_insert (node_t *q, node_t t);
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) * t \mapsto _ * \mathsf{a}(t)$

```
        mov ecx, [esp+4]
        mov edx, [esp+8]
        mov [edx+_next], NULL   // t->next = NULL;
        mov eax, [ecx]          // node_t p = *q;
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t \wedge \mathsf{eax} = \mathsf{head}(tl) \wedge \mathsf{ecx} = q \wedge \mathsf{edx} = t$

```
        cmp eax, NULL           // if (p == NULL) // add as first element
        jne qi_els
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t$
$\wedge \mathsf{eax} = \mathsf{head}(tl) = \mathsf{NULL} \wedge \mathsf{ecx} = q \wedge \mathsf{edx} = t$

```
                                // unfold, cast
```

Local: $[0]$; Pre: $q \mapsto \_ * \mathsf{a}\ t *$ node_t_ref$(\mathsf{a},[],t) \wedge \mathsf{ecx} = q \wedge \mathsf{edx} = t \wedge tl = []$

```
        mov [ecx], edx          // *q = t;
```

Local: $[0]$; Pre: $q \mapsto t * \mathsf{a}\ t *$ node_t_ref$(\mathsf{a},[],t) \wedge \mathsf{ecx} = q \wedge \mathsf{edx} = t \wedge tl = []$

```
                                // fold, cast
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl@[t],q)$

```
        ret                     // return;
```

```
qi_els:
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t$
$\wedge \mathsf{eax} = \mathsf{head}(tl) \neq \mathsf{NULL} \wedge \mathsf{ecx} = q \wedge \mathsf{edx} = t$

```
                                // cast
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t$
$\wedge \mathsf{eax} = \mathsf{last}(tl1) \wedge \mathsf{edx} = t \wedge tl = tl1@tl2$

```
        mov ecx, eax            // while (p->next != NULL) // insert at the tail
        mov eax, [eax+_next]    // p = p->next;
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t$
$\wedge \mathsf{eax} = \mathsf{head}(tl2) \wedge \mathsf{ecx} = \mathsf{last}(tl1) \wedge \mathsf{edx} = t \wedge tl = tl1@tl2$

```
        cmp eax, NULL
        jne qi_els
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t$
$\wedge \mathsf{eax} = \mathsf{head}(tl2) = \mathsf{NULL} \wedge \mathsf{ecx} = \mathsf{last}(tl1) \wedge \mathsf{edx} = t \wedge tl = tl1@tl2$

```
                                // cast
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl,q) *$ node_t_ref$(\mathsf{a},[],t) * \mathsf{a}\ t \wedge \mathsf{ecx} = \mathsf{last}(tl) \wedge \mathsf{edx} = t$

```
        mov [ecx+_next], edx    // p->next = t;
```

Local: $[0]$; Pre: node_t_ref$(\mathsf{a},tl@[t],q)$

```
        ret                     // return;
```

Figure 6.8: Verification of queue insertion

Fn q { Aux: tl, a; Local: $[0]$
Pre: $\text{node\_t\_ref}(\mathsf{a}, tl, q)$;
Post: $\text{node\_t\_ref}(\mathsf{a}, tl', , q) \wedge ((retv = \text{NULL} \wedge tl = tl' = [])$
$\vee (retv \neq \text{NULL} \wedge tl = retv :: tl' \wedge retv \mapsto \_ * \mathsf{a}\ retv))$ }

```
queue_delete:                          // node_t queue_delete (node_t *q);
```

Local: $[0]$; Pre: $\text{node\_t\_ref}(\mathsf{a}, tl, q)$

```
        mov ecx, [esp+4]
        mov eax, [ecx]          // node_t t = *q;
```

Local: $[0]$; Pre: $\text{node\_t\_ref}(\mathsf{a}, tl, q) \wedge \text{eax} = \text{head}(tl) \wedge \text{ecx} = q$

```
        cmp eax, NULL           // if (t != NULL)
        je qd_ret
```

Local: $[0]$; Pre: $\text{node\_t\_ref}(\mathsf{a}, tl, q) \wedge \text{eax} = \text{head}(tl) \wedge \text{ecx} = q \wedge \text{eax} \neq \text{NULL}$

```
                                // unfold, cast
```

Local: $[0]$; Pre: $q \mapsto t * \mathsf{a}\ t * \text{node\_t\_ref}(\mathsf{a}, tl', t) \wedge \text{eax} = t \wedge \text{ecx} = q \wedge \text{eax} \neq \text{NULL} \wedge tl = t :: tl'$

```
        mov edx, [eax+_next]    // *q = t->next;
        mov [ecx], edx
```

Local: $[0]$; Pre: $t \mapsto \_ * \mathsf{a}\ t * \text{node\_t\_ref}(\mathsf{a}, tl', q) \wedge \text{eax} = t \wedge \text{eax} \neq \text{NULL} \wedge tl = t :: tl'$

```
        ret                     // return t;

qd_ret:
```

Local: $[0]$; Pre: $\text{node\_t\_ref}(\mathsf{a}, tl, q) \wedge \text{eax} = \text{head}(tl) \wedge \text{ecx} = q \wedge \text{eax} = \text{NULL}$

```
        ret                     // return t;
```

Figure 6.9: Verification of queue deletion

Chapter 7

A Certified Mini Thread Library

In this chapter, we describe the mechanized verification of a thread implementation at machine level using XCAP86. Using the first mechanized proof for the safety of a machine-level thread implementation, we want to demonstrate the power and practicality of the XCAP framework, and that the certification of complex machine-level system code is not beyond reach. Our mini thread library, MTH, written in Mini86, is divided into modules for polymorphic queue (presented in the previous chapter), memory management, machine context, and threading.

All MTH code are at the same realistic level as the context switching example, following the Windows/Unix style. MTH and its verification involve neither change of system programming style nor disposal of existing system code base. There is no performance penalty, nor is any compatibility issue.

The specifications and proof of MTH modules and routines are modular. Each piece of code is specified and proved with minimal reference to external code. For example, in the verification of context module, there is no mentioning of thread at all.

We first give an overview of MTH structure, threading model, and threading features. Then we present the verifications of the machine context module and the threading module independently.

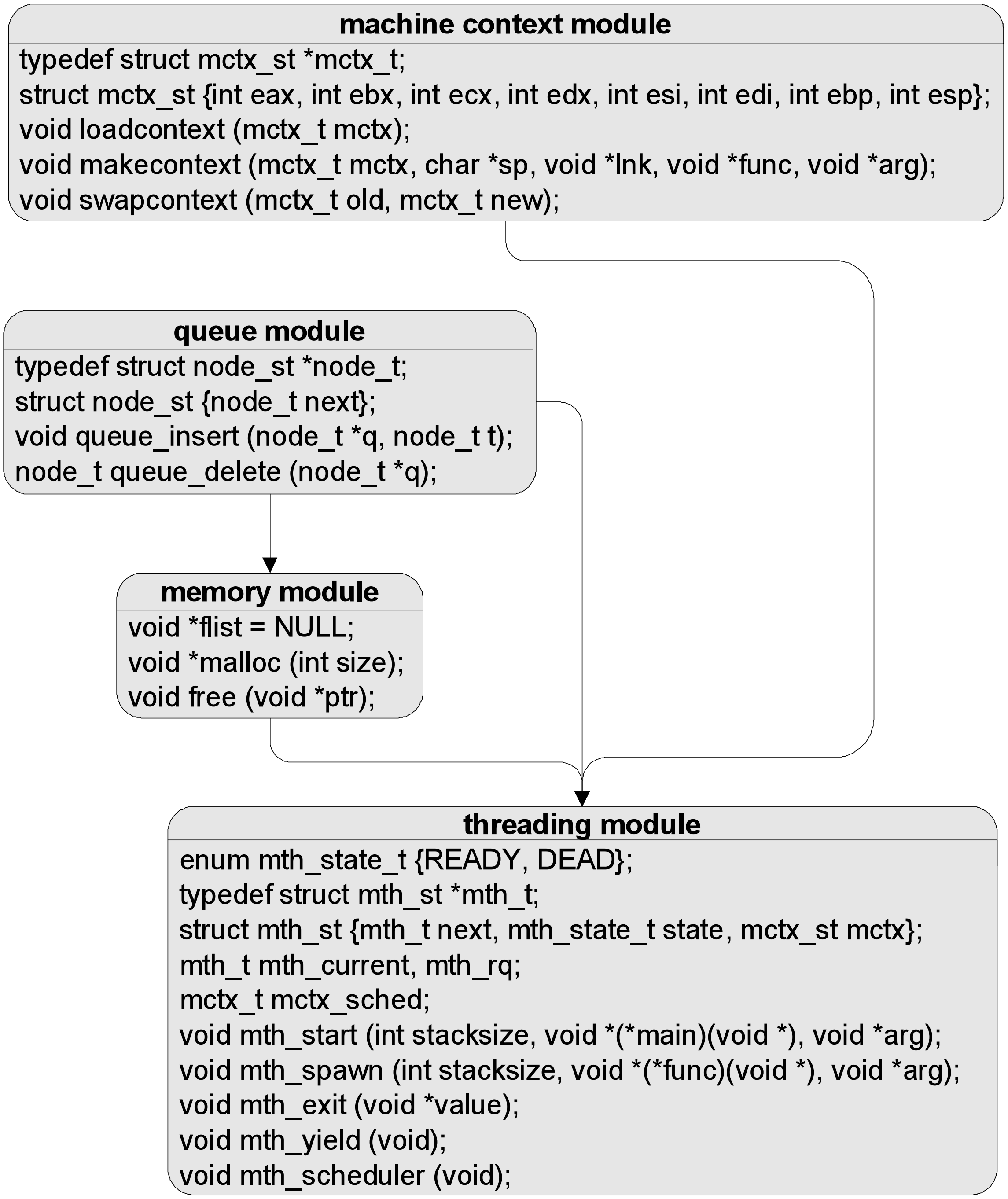
7.1 MTH: A Mini Thread Library

MTH is a minimal thread library initially modeled after the GNU Portable Threads (Pth) [17]. , a portable POSIX/ANSI-C based library for Unix. MTH supports non-preemptive scheduling for cooperative multi-threading. While sharing the same address space, every thread has its own program counter, register set, and stack. For simplicity, MTH omits some advanced features from Pth, including priority-based scheduling and events and signals, and adopts some small changes in the library structures. Nonetheless, the handling of machine contexts and thread management reflects sophisticated invariants of multi-threading and suffices in demonstrating the difficulty of verifying machine-level thread implementation.

MTH structures. Figure 7.1 divides MTH into several modules. Thread primitives visible to the user are implemented in the *threading* module, which refers to three other modules: *machine context*, *memory* and *queue*. The *queue* module is the one discussed in the previous chapter. Pseudo-code interfaces of these modules are provided for the ease of understanding. In later sections, we will develop more accurate specifications for verification purposes.

The memory module is a miniature library for dynamic storage allocation. It provides routines for heap allocation and deallocation. The queue module supports insertion and removal of queue elements in the expected way. Its routines are polymorphic and can be applied to different types of queues, as long as every queue element stores its link pointer in its first word.

The machine context module deserves some attention. A machine context (type mctx_st) stores the values of general-purpose registers, where the stack pointer `esp` should point to a stack with a valid return address on top. Context switching `swapcontext`() saves the current context as `old`, and loads a new one from `new` for further execution. Context loading `loadcontext`() loads a new context and executes from there. Context creation



```
typedef struct mctx_st *mctx_t;
struct mctx_st {int eax, int ebx, int ecx, int edx, int esi, int edi, int ebp, int esp};
void loadcontext (mctx_t mctx);
void makecontext (mctx_t mctx, char *sp, void *lnk, void *func, void *arg);
void swapcontext (mctx_t old, mctx_t new);
```

```
typedef struct node_st *node_t;
struct node_st {node_t next};
void queue_insert (node_t *q, node_t t);
node_t queue_delete (node_t *q);
```

```
void *flist = NULL;
void *malloc (int size);
void free (void *ptr);
```

```
enum mth_state_t {READY, DEAD};
typedef struct mth_st *mth_t;
struct mth_st {mth_t next, mth_state_t state, mctx_st mctx};
mth_t mth_current, mth_rq;
mctx_t mctx_sched;
void mth_start (int stacksize, void *(*main)(void *), void *arg);
void mth_spawn (int stacksize, void *(*func)(void *), void *arg);
void mth_exit (void *value);
void mth_yield (void);
void mth_scheduler (void);
```

Figure 7.1: Module structure and pseudo-C specification of MTH

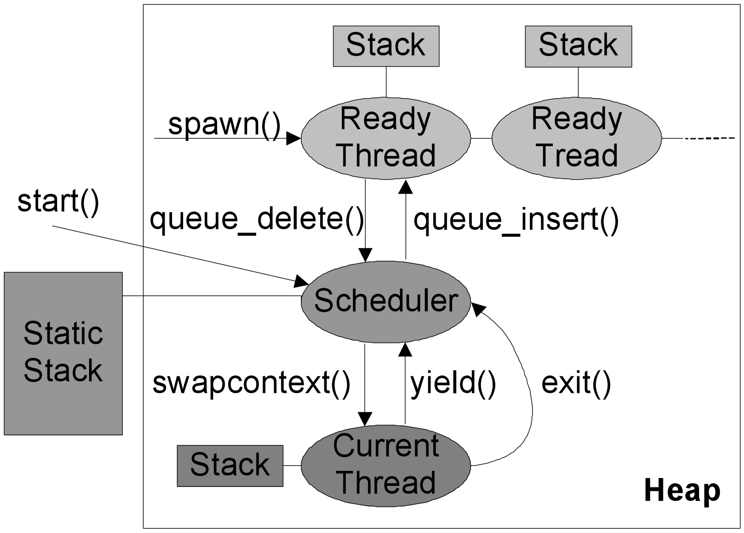


Figure 7.2: Threading model of MTH

`makecontext`() initializes a new context based on its arguments (location of new context, stack pointer, return link, address of target function, and argument for it); the stack pointer of the new context points to a stack frame prepared for the target function.

Threading model. The threading module provides interfaces for all the "visible" functionalities of MTH. The threading model is illustrated in Figure 7.2 When a new thread is *spawned*, it is put into a ready queue. A central scheduler removes a thread from the ready queue and makes it the *current thread* using context switching. The current thread may *exit* execution and become dead or *yield* execution and be put back into the ready queue.

A user program starts multi-threading by calling *mth_start*(), which will set up the threading data structures, spawn the first user thread, and call *mth_scheduler*() to initiate the scheduling. All these happen on the statically allocated system stack, which is different from the thread stacks to be allocated on the heap. The scheduler is essentially a loop implementing simple FIFO scheduling.

The thread creation function *mth_spawn*() takes three arguments describing the new thread: the stack size, the location of the thread code, and the thread arguments. It allocates a thread stack and a thread control block (TCB), creates a machine context for the thread, and puts the TCB into the ready queue. Once made current (or running), a thread's execution will not be interrupted unless it explicitly yields control using *mth_yield*(), which transfers the control to the scheduler using context switching. When the main function of a thread returns, *mth_exit* is invoked to mark the thread as dead and yield control to the scheduler. Then, the scheduler will reclaim the resources and terminate the dead thread.

Machine-level thread implementations like MTH pose many challenges for verification. The complex and subtle behaviors of the routines are difficult to model. The control flow is complex with the extensive use of embedded code pointers stored in machine contexts and TCBs. Furthermore, the invariants maintained by the threads are recursive by nature, because the threads have mutual expectations from each other.

The memory module. The memory module is self-contained, as described by the specification below. It makes use of a free list ($flist$) of memory blocks implemented using the queue data type. Its verification is adapted from that of a malloc/free library by Yu *et al.* [63]. We omit the details and instead only point out that there is a memory invariant $\mathsf{I}_{\mathsf{mem}}$ (essentially stating the existence of the free list) that must be maintained during the execution of client programs of this module. Its use will be illustrated in later sections.

$$\begin{aligned}
\mathsf{mblk\_t}\ t &\triangleq \exists size.\ t{-}4 \mapsto size \ast t{+}4 \mapsto [size{-}4] \wedge size \geq 4 \\
\mathsf{I}_{\mathsf{mem}} &\triangleq \exists tl.\ \mathsf{node\_t\_ref}(\mathsf{mblk\_t}, tl, flist)
\end{aligned}$$

$$\begin{aligned}
malloc &: \mathsf{Fn}\ \ size\ \{\mathsf{Aux}::;\ \ \mathsf{Local}:\ [28]; \\
\mathsf{Pre} &: \mathsf{I}_{\mathsf{mem}}; \\
\mathsf{Post} &: \mathsf{I}_{\mathsf{mem}} \ast retv{-}4 \mapsto size \ast retv \mapsto [size]\}
\end{aligned}$$

$$\begin{aligned}
free &: \mathsf{Fn}\ \ ptr\ \{\mathsf{Aux}::;\ \ \mathsf{Local}:\ [8]; \\
\mathsf{Pre} &: \mathsf{I}_{\mathsf{mem}} \ast ptr{-}4 \mapsto size \ast ptr \mapsto [size]; \\
\mathsf{Post} &: \mathsf{I}_{\mathsf{mem}}\}
\end{aligned}$$

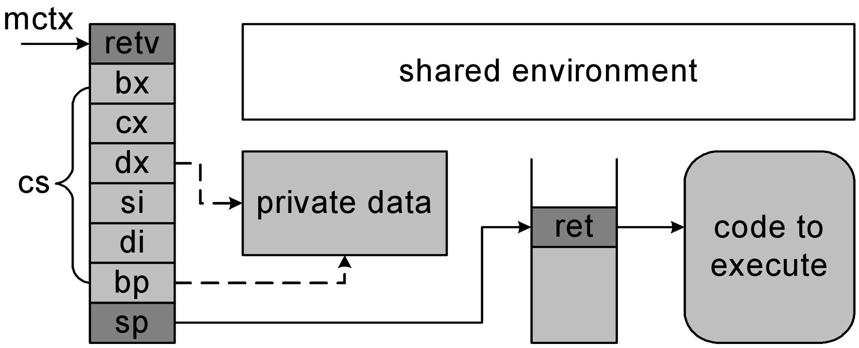


Figure 7.3: Machine context

7.2 Verification of the Machine Context Module

Machine context is a primitive concept for low-level programming. It offers the basis for higher-order control-flow transfers, such as call/cc and threading. As is the case of the queue and memory modules, the machine context module is specified and verified separately without any knowledge about its client—the threading module.

What is a machine context? Although the pseudo-C specification in Figure 7.1 seems to indicate that a context is merely eight words, the actual assumptions behind it are rather complex. As illustrated in Figure 7.3, the eight words represent a return value *retv*, six registers referred to collectively as *cs*, (the *cs* convention for context is different than that of , there are six of them, instead of the normal four), and a stack pointer *sp*. *sp* points to a return address *ret* typically found on top of a stack frame. (A machine context does not really care about other part of the stack, or if there is a stack at all). The six saved registers may point to some private data. (which could contains the other part of the stack), There may also be some environment data shared with external code. Eventually, a context is consumed when invoked. A proper invocation by jumping to the return address *ret* requires (1) the saved register contents be restored into the register file, (2) the stack and private data be preserved, and (3) the shared environment be available. In other words, it is only safe to load a context when its global shared data is available.

All these requirements make it difficult to specify and verify the seemingly simple manipulations on machine contexts. To facilitate the reasoning, we define a macro for the context data structure $\mathsf{mctx\_t}(\mathsf{a}_{env}, mctx)$, parametric to the environment described as a_{env}.

$$\begin{aligned} &\exists retv, cs, sp, ret, \mathsf{a}_{prv}. \qquad\qquad\qquad\qquad\qquad\qquad mctx \mapsto retv, cs, sp \ * \ sp \mapsto ret \ * \ \mathsf{a}_{prv} \\ &\quad \wedge \ \mathsf{cptr}(ret, \mathsf{reg}_6(cs, sp{+}4) \ \wedge \ \mathsf{eax} = retv \ \wedge \ \mathsf{a}_{env} \ * \ mctx \mapsto retv, cs, sp \ * \ sp \mapsto ret \ * \ \mathsf{a}_{prv}) \end{aligned}$$

where a_{prv} describes the private data, and reg_6 describes the 6 saved registers and esp:

$$\begin{aligned} \mathsf{reg}_6(ebx, ecx, edx, esi, edi, ebp, esp) \ \triangleq \ & \mathsf{ebx} = ebx \wedge \mathsf{ecx} = ecx \wedge \mathsf{edx} = edx \\ & \wedge \mathsf{esi} = esi \wedge \mathsf{edi} = edi \wedge \mathsf{ebp} = ebp \wedge \mathsf{esp} = esp \end{aligned}$$

Context switching. *swapcontext*() expects two pointers (old and new contexts) as arguments and performs three tasks: to save registers to old context, to load registers from new context, and to transfer control to new context. Observations from inside and outside of *swapcontext*() are very different. From the code of *swapcontext*(), it gets called by one client and "returns" to another. However, this is transparent to the clients—when *swapcontext*() returns to a client, the stack and private data of the client are kept intact.

The specification and proof outline of *swapcontext*() is given in Figure 7.4. It uses a macro Fn_6, a variant of Fn obtained by replacing reg with reg_6 and changing cs from referring to 4 registers to 6. Similarly to Fn, Fn_6 helps to manage the preservation of the stack and private data. In addition, the pre-condition of the specification dictates the shape of three pieces of memory: (1) the old context pointed to by *old*—at the beginning of the routine it is simply 32 bytes of memory available for use, (2) the shared data a_{env}, and (3) the new context pointed to by *new*. The new context is specified with help of the macro $\mathsf{mctx\_t}$. The environment parameter of this macro consists of two parts: the shared data a_{env} and another $\mathsf{mctx\_t}$ macro describing the old context. This is because the old context will be properly set up by the routine before switching to the new one. A tricky point is that the old context will be expecting a new (existentially quantified) shared environment a_{newenv}. Although some may expect the new environment to be simply the old shared a_{env} together with the new context at *new*, this may not necessarily be the case. For instance, the new context may not be still alive when the old context regains control. The post-condition of *swapcontext*() is relatively simple: the space for the old context would be available together with the new shared data a_{newenv}.

An interesting proof step is the one after the old context is packed but before the new one is unpacked. At this point, there is no direct notion of stack or function. The relevant machine state essentially comprises of two contexts and one environment:

$$\begin{aligned} \mathsf{eax} = new \ \wedge \quad & \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env} \\ & * \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new) \end{aligned}$$

It is therefore safe to load the new context from eax and switch to it. More proof steps are available in Figure 7.4.

Context loading. *loadcontext*() essentially performs half of the tasks of *swapcontext*(), with some slight difference in the stack layout. Although we call it a "function", it actually never returns and does not require the stack top to be a valid return address. Instead of using Fn macro, we only write the following as its complete pre-condition. We present the verification of *loadcontext*() in Figure 7.5.

Context creation. We present the specification and proof outline of *makecontext*() in Figure 7.6. The intermediate assertions are organized using the Fn macro. For conciseness, we omitted common parts of these macros, thus emphasizing only the changing parts: the stack frame Local and the current pre-condition Pre.

The pre-condition of the routine specifies (1) an empty context at *mctx*, (2) a stack *nsp* with available space of size *nss*, (3) some private data of the target context (potentially accessible from the argument *arg* of the function $func$() of the target context, (4) a link (return address) *lnk* to be used when the target context finishes execution, and (5) a function pointer $func$() for the code to be executed in the target context. It also specifies the exact requirements on the code at pointers *lnk* and $func$: (1) a_{ret} occurs both in the post-condition of $func$() and in the pre-condition of *lnk*, indicating that the code at the return address *lnk* may expect some results from the context function $func$(); (2) the stack should

$$\begin{array}{l}
\mathsf{Fn}_6\ old, new\ \{\ \mathsf{Aux:}\ \mathsf{a}_{newenv};\ \ \mathsf{Local:}\ [0];\\
\quad \mathsf{Pre:}\ \ old \mapsto [32] * \mathsf{a}_{env} * \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new);\\
\quad \mathsf{Post:}\ old \mapsto [32] * \mathsf{a}_{newenv} \wedge \mathsf{eax} = 0\ \}
\end{array}$$

```
swapcontext:                        // void swapcontext (mctx_t old, mctx_t new);
```

$$\begin{array}{l}
\qquad\quad \mathsf{reg}_6(cs, sp) \quad \wedge ss \geq 0 \quad \wedge \mathsf{stack}(sp, ss, ret, old, new) \quad * old \mapsto [32] \quad * \mathsf{a}_{prv} * \mathsf{a}_{env}\\
* \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new)\\
\wedge \mathsf{cptr}(ret, \mathsf{reg}_6(cs, sp+4) \wedge \mathsf{eax} = 0 \wedge \mathsf{stack}(sp+4, ss+4, old, new) * old \mapsto [32] \quad * \mathsf{a}_{prv} * \mathsf{a}_{newenv})
\end{array}$$

```
        mov eax, [esp+4]     // load address of the context data structure we save in
        mov [eax+_eax], 0    // all registers preserved except eax
        mov [eax+_ebx], ebx
        mov [eax+_ecx], ecx
        mov [eax+_edx], edx
        mov [eax+_esi], esi
        mov [eax+_edi], edi
        mov [eax+_ebp], ebp
        mov [eax+_esp], esp
        mov eax, [esp+8]     // load address of the context data structure we have to load
```

$$\begin{array}{l}
\qquad\quad \mathsf{eax} = new \qquad \wedge ss \geq 0 \quad \wedge \mathsf{stack}(sp, ss, ret, old, new) \quad * old \mapsto 0, cs, sp * \mathsf{a}_{prv} * \mathsf{a}_{env}\\
* \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new)\\
\wedge \mathsf{cptr}(ret, \mathsf{reg}_6(cs, sp+4) \wedge \mathsf{eax} = 0 \wedge \mathsf{stack}(sp+4, ss+4, old, new) * old \mapsto [32] \quad * \mathsf{a}_{prv} * \mathsf{a}_{newenv})
\end{array}$$

```
                                    // shuffle and cast
```

$$\begin{array}{l}
\qquad\qquad \mathsf{eax} = new\ \wedge \qquad old \mapsto 0, cs, sp * sp \mapsto ret * \mathsf{stack}(sp, ss) * sp+4 \mapsto old, new * \mathsf{a}_{prv}\\
\wedge \mathsf{cptr}(ret, \mathsf{reg}_6(cs, sp+4)\\
\qquad\quad \wedge \mathsf{eax} = 0 \wedge \mathsf{a}_{newenv} * old \mapsto 0, cs, sp * sp \mapsto ret * \mathsf{stack}(sp, ss) * sp+4 \mapsto old, new * \mathsf{a}_{prv})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad * \mathsf{a}_{env} * \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new)
\end{array}$$

```
                                    // pack old context
```

$$\mathsf{eax} = new\ \wedge \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env} * \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new)$$

```
                                    // unpack new context
```

$$\begin{array}{l}
\qquad\qquad \mathsf{eax} = new\ \wedge \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env} * new \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{newprv}\\
\wedge \mathsf{cptr}(ret', \mathsf{reg}_6(cs', sp'+4)\\
\qquad\qquad \wedge \mathsf{eax} = retv' \wedge \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env} * new \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{newprv})
\end{array}$$

```
        mov esp, [eax+_esp] // load the new stack pointer.
        mov ebp, [eax+_ebp]
        mov edi, [eax+_edi]
        mov esi, [eax+_esi]
        mov edx, [eax+_edx]
        mov ecx, [eax+_ecx]
        mov ebx, [eax+_ebx]
        mov eax, [eax+_eax]
```

$$\begin{array}{l}
\ \mathsf{reg}_6(cs', sp') \wedge \mathsf{eax} = retv' \wedge \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env} * new \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{newprv}\\
\wedge \mathsf{cptr}(ret', \mathsf{reg}_6(cs', sp'+4)\\
\qquad\qquad \wedge \mathsf{eax} = retv' \wedge \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env} * new \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{newprv})
\end{array}$$

```
        ret
```

Figure 7.4: Verification of machine context switching

```
loadcontext:                       // void loadcontext (mctx_t mctx);
$\mathsf{reg}(cs,sp) \wedge \mathsf{stack}(sp,ss,ret,mctx) * \mathsf{a}_{env} * \mathsf{mctx\_t}(\mathsf{stack}(sp,ss,ret,mctx) * \mathsf{a}_{env}, mctx)$
        mov eax, [esp+4]
$\mathsf{eax} = mctx \wedge \mathsf{stack}(sp,ss,ret,mctx) * \mathsf{a}_{env} * \mathsf{mctx\_t}(\mathsf{stack}(sp,ss,ret,mctx) * \mathsf{a}_{env}, mctx)$
                                   // unpack context
$\mathsf{eax} = mctx \qquad \wedge\, \mathsf{a}_{newenv} * \mathsf{mctx\_t}(\mathsf{a}_{newenv}, mctx)$
                                   // unpack context
$\mathsf{eax} = mctx \qquad \wedge\, \mathsf{a}_{newenv} * mctx \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{prv}$
$\wedge \mathsf{cptr}(ret', \mathsf{reg}_6(cs', sp'+4) \wedge \mathsf{eax} = retv' \wedge \mathsf{a}_{newenv} * mctx \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{prv})$
        mov esp, [eax+_esp] // load the new stack pointer.
        mov ebp, [eax+_ebp]
        mov edi, [eax+_edi]
        mov esi, [eax+_esi]
        mov edx, [eax+_edx]
        mov ecx, [eax+_ecx]
        mov ebx, [eax+_ebx]
        mov eax, [eax+_eax]
   $\mathsf{reg}_6(cs', sp'+4) \wedge \mathsf{eax} = retv' \wedge \mathsf{a}_{newenv} * mctx \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{prv}$
$\wedge \mathsf{cptr}(ret', \mathsf{reg}_6(cs', sp'+4) \wedge \mathsf{eax} = retv' \wedge \mathsf{a}_{newenv} * mctx \mapsto retv', cs', sp' * sp' \mapsto ret' * \mathsf{a}_{prv})$
        ret
```

Figure 7.5: Verification of machine context loading

Fn $mctx, nsp, lnk, func, arg\ \{$ Aux: a_{env}; Local: $[0]$;
Pre: $mctx \mapsto [32] * \mathsf{stack}(nsp, nss) * \mathsf{a}_{prv} \wedge nss \geq 12$
$\wedge\mathsf{cptr}(lnk,\ \mathsf{esp} = nsp-4 \wedge \mathsf{stack}(nsp-4, nss-4, arg) * \mathsf{a}_{ret})$
$\wedge\mathsf{cptr}(func,$ Fn $arg'\ \{$ Local: $nss-8$;
Pre: $sp' = nsp - 8 \wedge arg' = arg \wedge \mathsf{a}_{env} * mctx \mapsto [32] * \mathsf{a}_{prv}$;
Post: $\mathsf{a}_{ret}\ \});$
Post: $\mathsf{mctx\_t}(\mathsf{a}_{env}, mctx)\ \}$

```
makecontext:   // void makecontext (mctx_t mctx, char *sp, void *lnk, void *func, void *arg);
```

Local: $[0]$; Pre: $mctx \mapsto [32] * \mathsf{stack}(nsp, nss) * \mathsf{a}_{prv} \wedge nss \geq 12$
$\wedge\mathsf{cptr}(lnk,\ \mathsf{esp} = nsp-4 \wedge \mathsf{stack}(nsp-4, nss-4, arg) * \mathsf{a}_{ret})$
$\wedge\mathsf{cptr}(func,$ Fn $arg'\ \{$ Local: $nss-8$;
Pre: $sp' = nsp - 8 \wedge arg' = arg \wedge \mathsf{a}_{env} * mctx \mapsto [32] * \mathsf{a}_{prv}$;
Post: $\mathsf{a}_{ret}\ \});$

```
        mov eax, [esp+4]      // load address of the context data structure.
        mov ecx, [esp+8]      // load new stack top
        mov edx, [esp+20]     // push the function's argument into new stack
        mov [ecx-4], edx
        mov edx, [esp+12]     // push link (return address for the function) into new stack
        mov [ecx-8], edx
        mov edx, [esp+16]     // push the function as return IP into new stack
        mov [ecx-12], edx
        sub ecx, 12
        mov [eax+_esp], ecx   // only the stack pointer matters for a fresh new context
```

Local: $[0]$; Pre: $mctx \mapsto [28], nsp-12 * \mathsf{stack}(nsp-12, nss-12, func, lnk, arg) * \mathsf{a}_{prv}$
$\wedge\mathsf{cptr}(lnk,\ \mathsf{esp} = nsp-4 \wedge \mathsf{stack}(nsp-4, nss-4, arg) * \mathsf{a}_{ret})$
$\wedge\mathsf{cptr}(func,$ Fn $arg'\ \{$ Local: $nss-8$;
Pre: $sp' = nsp - 8 \wedge arg' = arg \wedge \mathsf{a}_{env} * mctx \mapsto [32] * \mathsf{a}_{prv}$;
Post: $\mathsf{a}_{ret}\ \});$

// unfold, shuffle, and cast

Local: $[0]$; Pre: $mctx \mapsto retv', cs', nsp-12 * nsp-12 \mapsto func$
$* \mathsf{stack}(nsp-12, nss-12) * nsp-8 \mapsto lnk, arg * \mathsf{a}_{prv}$
$\wedge\mathsf{cptr}(lnk,\ \mathsf{esp} = nsp-4 \wedge \mathsf{stack}(nsp-4, nss-4, arg) * \mathsf{a}_{ret})$
$\wedge\mathsf{cptr}(func,\ \mathsf{reg}_6(cs', nsp-8) \wedge \mathsf{eax} = retv' \wedge \mathsf{a}_{env} * mctx \mapsto retv', cs', nsp-12 * nsp-12 \mapsto func$
$* \mathsf{stack}(nsp-12, nss-12) * nsp-8 \mapsto lnk, arg * \mathsf{a}_{prv}$
$\wedge\mathsf{cptr}(lnk,\ \mathsf{esp} = nsp-4 \wedge \mathsf{stack}(nsp-4, nss-4, arg) * \mathsf{a}_{ret}));$

// pack the fresh context

Local: $[0]$; Pre: $\mathsf{mctx\_t}(\mathsf{a}_{env}, mctx)$;

```
        ret
```

Figure 7.6: Verification of machine context creation

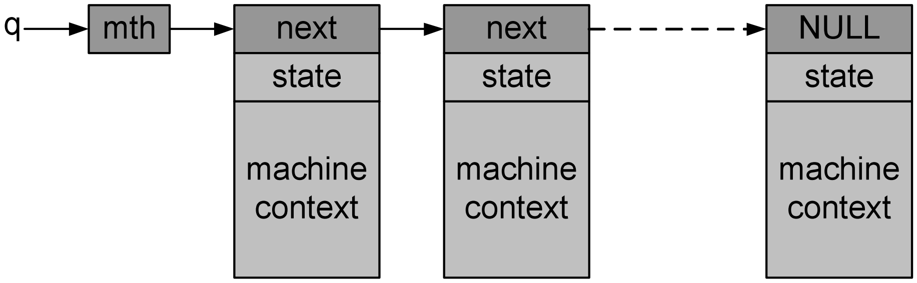


Figure 7.7: Thread control blocks and queues of MTH

be properly maintained upon returning to lnk.

The post-condition of the routine simply states that, when $makecontext()$ returns, $mctx$ will point to a proper context where the shared environment is a_{env}.

Our interface of $makecontext()$ is faithful to the Unix/Linux implementations. The heavy usage of embedded code pointers and function pointers in this case shows how crucial XCAP's support of ECP is. As the proof shows, the most complex step is again on casting code pointers' pre-conditions and pack all the resource into a complete context.

7.3 Verification of the Threading Module

With all the other modules verified, we are now ready to tackle the threading module.

Thread control blocks and thread queues. The main data structure for MTH threading is the thread control block (TCB). Figure 7.7 presents a queue of TCB's. Every TCB comprises of three parts: (1) the first word is a link field for the queue, (2) the second word stores the status of the thread (READY or DEAD), and (3) a machine context is embedded in the next eight words. The TCB's of all ready but non-running threads are put into a ready queue, implemented as an instance of the polymorphic queue introduced in Section 6.3.

$$\mathsf{mth\_t\_ref}(st, \mathsf{a}_{inv}, tl, q) \triangleq \mathsf{node\_t\_ref}(\mathsf{mth\_t}(st, \mathsf{a}_{inv}), tl, q).$$

As a reminder, node_t_ref requires three parameters: a predicate describing the queue nodes, a list of node locations, and a queue pointer. mth_t_ref requires 4 parameters: (1)

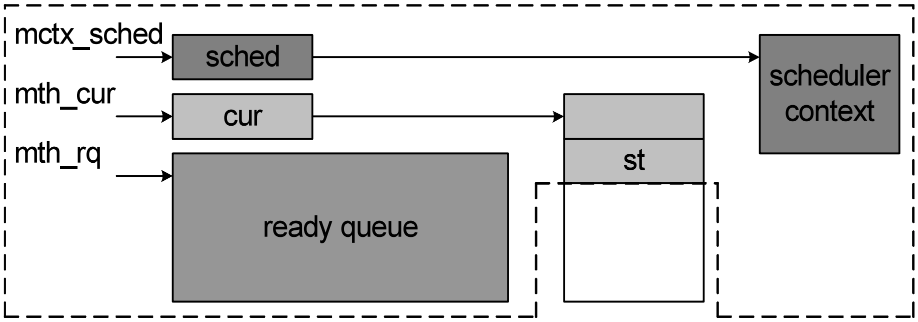


Figure 7.8: Threading invariant of MTH

every thread in the queue has the same thread state st, (2) a certain threading invariant a_{inv} that must be established before any thread in the queue takes control, (3) a list tl of TCB handles in the queue, and (4) the queue pointer. Note that there should be a threading invariant expected by each thread's context when being switched to. Since this invariant is not yet known, we name it as a_{inv}. These arguments are organized properly to feed the $\mathsf{node\_t\_ref}$ macro. In particular, the content of a TCB (excluding the link field) must satisfy $\mathsf{mth\_t}(st, \mathsf{a}_{inv})$, which is defined as:

$$\lambda mth.\ mth{+}4 \mapsto st * \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{a}_{inv}(mth), mth{+}8)$$

In short, a TCB at mth satisfies the $\mathsf{mth\_t}$ macro if the state st is located at offset 4 and a valid context is located at offset 8. Besides the threading invariant a_{inv} (to be provided by the current thread), the context also expects the invariant $\mathsf{I}_{\mathsf{mem}}$ of the memory module.

Threading invariant. Figure 7.8 illustrates the run-time threading data structures and invariants. There are three entities involved: a scheduler context, a TCB of the running thread, and a ready queue, each pointed to by a global memory address. When a thread yields or exits, the scheduler context is invoked to do the scheduling.

A thread expects such an environment to be properly maintained before taking control. Meanwhile, itself is also part of such kind of environment for other threads. Therefore, the threading invariant is inherently recursive, as modelled in the following:

$$
\begin{aligned}
\mathsf{I}_{\mathsf{core}} \triangleq \mu\alpha.\,\lambda st, cur.\ \exists sched, tl.\ & \\
& \mathsf{mctx\_sched} \mapsto sched \ * \ \mathsf{mctx\_t}(\mathsf{sched\_env}(\alpha), sched) \\
* \ & \mathsf{mth\_cur} \mapsto cur \ * \ cur \mapsto \_, st \\
* \ & \mathsf{mth\_t\_ref}(\mathsf{READY}, \alpha(\mathsf{READY}), tl, \mathsf{mth\_rq})
\end{aligned}
$$

The last three lines correspond to the three items in Figure 7.8, respectively. The recursive variable α essentially represents $\mathsf{I}_{\mathsf{core}}$ itself, which is exactly the threading invariant that every TCB expects. Note that this invariant only describes the parts inside the dashed lines in Figure 7.8—the context of the current thread is not part of its own shared environment (nevertheless, each thread's context considers all other threads' contexts as part of its shared environment).

The environment of the scheduler's context, described above using sched_env, deserves further explanation. We use the macro $\mathsf{sched\_env}(\mathsf{a}_{inv})$ to represent:

$$
\begin{aligned}
\exists st, cur.\ \mathsf{I}_{\mathsf{mem}} \ * \ & \mathsf{mctx\_sched} \mapsto sched \\
* \ & \mathsf{mth\_cur} \mapsto cur \ * \ cur \mapsto \_, st \\
* \ & \mathsf{mth\_t\_ref}(\mathsf{READY}, \mathsf{a}_{inv}(\mathsf{READY}), tl, \mathsf{mth\_rq}) \\
* \ & ((st = \mathsf{READY} \ \wedge \ \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} \ * \ \mathsf{a}_{inv}(\mathsf{READY}, cur), cur+8)) \ \vee \\
& \ (st = \mathsf{DEAD} \quad \wedge \ \exists size.\ cur-4 \mapsto size \ * \ cur+8 \mapsto [size-8])).
\end{aligned}
$$

The first three lines describe the existence of the scheduler pointer (not the scheduler context), the running thread, and the ready queue. When the scheduler takes control, there must have been a user thread that yielded. The last two lines describe the context of that yielding thread—based on whether the thread is still alive as indicated by the state field *st*, there would be either a proper context or the corresponding space available.

The $\mathsf{I}_{\mathsf{core}}$ macro is instantiated with the READY state and a running thread before used as the invariant of the threading module:

$$
\mathsf{I}_{\mathsf{full}}(cur) \triangleq \mathsf{I}_{\mathsf{core}}(\mathsf{READY}, cur) \ * \ cur+8 \mapsto [32]
$$

The *cur* pointer of the running thread is irrelevant to the clients of the threading module. The external invariant used when importing the threading module is simply

$$\mathsf{I}_{\mathsf{mth}} \triangleq \mathsf{I}_{\mathsf{full}}(\_)$$

where _ is a wildcard. We also define a syntactic sugar to describe the ready queue:

$$\mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) \ \triangleq \ \mathsf{mth\_t\_ref}(\mathsf{READY}, \mathsf{I}_{\mathsf{core}}(\mathsf{READY}), tl, \mathsf{mth\_rq}).$$

Thread yielding. The most frequently used threading routine should be *mth_yield*(), implemented as a context switch from the current (running) thread to the scheduler. Figure 7.9 gives its specification and proof outline. Similar to *swapcontext*(), Fn_6 is used here. After unpacking the threading invariant, pointers to the contexts of the current thread and the scheduler are pushed onto the stack. A call to *swapcontext*() completes the yielding. As indicated in the assertions, the thread's context expects both the memory and the threading invariants to be maintained before it takes control again. Interested readers could refer to Figure 7.4 to better understand the verification.

Thread creation. Dynamic thread creation is also a basic requirement for thread implementations. Just as *mth_yield*() relies on *swapcontext*() to perform switching, *mth_spawn*() creates a new thread with the help of *makecontext*(). When creating a new thread, one should specify the size (*stacksize*) of the thread stack, a pointer (*func*) to the thread code, and an argument pointer (*arg*) for the thread code. The data at *arg* is described by an existentially quantified predicate a_{pre} that describes the private data prepared for *func*(). *mth_spawn*() will call *malloc*() to allocate a big chunk of memory, divide it into the TCB and the stack of the new thread, initialize TCB, call *makecontext*() to build the new thread context, and call *queue_insert*() to append the new thread onto the ready queue. The verification of *mth_spawn*() is presented in Figure 7.10 and Figure 7.11.

The pre- and post-conditions of *mth_spawn*() require that the memory and threading invariants be maintained, which is necessary for verifying the memory allocation and threading operations in the code. The pre- and post-conditions of *func*(), as asserted with cptr, also maintain the same invariants. In addition, *func*() is to be invoked with the given argument *arg*. The post-condition of *func*() involves the invariants only. This indicates

Fn_6 $\{$ Aux: ; Local : $[12]$;
Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$;
Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ $\}$

`mth_yield:` *// void mth_yield (void);*

Local: $[12]$; Pre: $cur{+}8 \mapsto [32]$
$* \mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$* \mathsf{mctx\_t}(\mathsf{sched\_env}(\mathsf{I}_{\mathsf{core}}), sched)$

```
    mov eax, [mctx_sched] // eax not preserved
    push eax
    mov eax, [mth_cur]
    add eax, 8
    push eax
```

Local: $[4], cur{+}8, sched$; Pre: $cur{+}8 \mapsto [32]$
$* \mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$* \mathsf{mctx\_t}(\mathsf{sched\_env}(\mathsf{I}_{\mathsf{core}}), sched)$

// unfold, shuffle, cast

Local: $[4], cur{+}8, sched$; Pre: $cur{+}8 \mapsto [32]$
$* \mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$* \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$* \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{core}}(\mathsf{READY}, cur), cur{+}8), sched)$

```
    call swapcontext // swapcontext(&mth_cur->mctx, mctx_sched);
```

Local: $[4], cur{+}8, sched$; Pre: $cur{+}8 \mapsto [32] * \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{core}}(\mathsf{READY}, cur)$

```
    add esp, 8
```

Local: $[12]$; Pre: $cur{+}8 \mapsto [32] * \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{core}}(\mathsf{READY}, cur)$

// fold, shuffle, cast

Local: $[12]$; Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$

```
    ret
```

Figure 7.9: Verification of thread yielding

Fn $stacksize, func, arg$ { Aux: ; Local : $[56]$;
  Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre} \wedge stacksize \geq 12 \wedge \mathsf{cptr}(func,$ Fn arg' { Local : $[stacksize-8]$;
    Pre: $arg' = arg \wedge \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre}$;
    Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ });
  Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ }

// cast (above is external interface, below is internal interface)

Fn $stacksize, func, arg$ { Aux: tl; Local : $[56]$;
  Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) * \mathsf{a}_{pre} \wedge stacksize \geq 12$
    $\wedge\ \mathsf{cptr}(func,$ Fn arg' { Local : $[stacksize-8]$;
    Pre: $arg' = arg \wedge \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre}$;
    Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ });
  Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl@[retv], \mathsf{mth\_rq})$ }

```
mth_spawn:                  // mth_t mth_spawn (int stacksize, void *(*func)(void *), void *arg);
```

Local: $[56]$; Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) * \mathsf{a}_{pre} \wedge stacksize \geq 12$
    $\wedge\ \mathsf{cptr}(func,$ Fn arg' { Local : $[stacksize-8]$;
    Pre: $arg' = arg \wedge \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre}$;
    Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ });

```
        mov ecx, [esp+8]
        add ecx, size_of_mth_st
        push ecx                // allocate a new thread control block
```

Local: $[52], 40+stacksize$; Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) * \mathsf{a}_{pre} \wedge stacksize \geq 12$
    $\wedge\ \mathsf{cptr}(func,$ Fn arg' { Local : $[stacksize-8]$;
    Pre: $arg' = arg \wedge \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre}$;
    Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ });

```
        call malloc             // mth_t t = (mth_t)malloc(sizeof(mth_st)+stacksize);
```

Local: $[52], 40+stacksize$; Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) * \mathsf{a}_{pre} * t-4 \mapsto 40+stacksize$
    $* \ t \mapsto [40+stacksize] \wedge \mathsf{eax} = t \wedge stacksize \geq 12$
    $\wedge\ \mathsf{cptr}(func,$ Fn arg' { Local : $[stacksize-8]$;
    Pre: $arg' = arg \wedge \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre}$;
    Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}$ });

```
        pop ecx
        mov [eax+_state], READY // t->state = READY;
        mov edx, [esp+16]
        push edx
        mov edx, [esp+12]
        push edx
        push mth_exit
        add ecx, eax
        push ecx
        add eax, _mctx
        push eax
```

Figure 7.10: Verification of thread creation

Local: $[36], t+8, t+40+stacksize, \mathsf{mth\_exit}, func, arg;$

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) * \mathsf{a}_{pre} * t-4 \mapsto 40+stacksize * t \mapsto \_, \mathsf{READY} * t+8 \mapsto [32]$
$\quad * \mathsf{stack}(t+40+stacksize, stacksize) \wedge stacksize \geq 12$
$\quad \wedge \mathsf{cptr}(\mathsf{mth\_exit},\ \mathsf{esp} = sp \wedge \mathsf{stack}(sp, ss) * \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{full}}(cur) * cur-4 \mapsto size$
$\qquad\qquad * sp \mapsto [size - 40 - ss] \wedge ss = sp - cur - 40)$
$\quad \wedge \mathsf{cptr}(func,\ \mathsf{Fn}\ arg' \ \{\ \mathsf{Local:}\ [stacksize-8];$
$\qquad\qquad \mathsf{Pre:}\ arg' = arg \wedge \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}} * \mathsf{a}_{pre};$
$\qquad\qquad \mathsf{Post:}\ \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{mth}}\ \});$

```
        call makecontext   // makecontext(&t->mctx, t+sizeof(mth_st)+stacksize,
                           //                                mth_exit,func,arg);
```

Local: $[36], t+8, t+40+stacksize, \mathsf{mth\_exit}, func, arg;$

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) * t \mapsto \_, \mathsf{READY} * \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{core}}(\mathsf{READY}, t), t+8)$

```
        pop eax
        add esp, 16
        sub eax, _mctx
        push eax
        push mth_rq
```

Local: $[48], \mathsf{mth\_rq}, t;$

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq}) \wedge t \mapsto \_ * \mathsf{mth\_t}(\mathsf{READY}, \mathsf{I}_{core}(\mathsf{READY}), t)$

```
        call queue_insert  // queue_insert(&mth_rq, t);
```

Local: $[48], \mathsf{mth\_rq}, t;$ Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl @ [t], \mathsf{mth\_rq})$

```
        add esp, 4
        pop eax
```

Local: $[56];$ Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{ready\_queue}(tl @ [t], \mathsf{mth\_rq})$

```
        ret
```

Figure 7.11: Verification of thread creation (continued)

that the data of a_{pre} must be properly deallocated before $func$() finishes, leaving no possibility of space leaks. Finally, the post-condition of *mth_spawn*() does not explicitly refer to the newly spawned thread, which would be in the ready queue as part of $\mathsf{I}_{\mathsf{mth}}$.

Thread termination. *mth_exit* is a routine that *mth_spawn*() supplies as the return link for newly created contexts. It is invoked when a thread finishes execution and returns from its main function. The verification of *mth_exit* is presented in Figure 7.12. The code simply sets the state of the current thread to be DEAD and transfers the control to the scheduler with $loadcontext$(). Since it is not a function, we can not use Fn syntax to present its pre-condition. The proof involves the common packing and unpacking of the thread-

```
mth_exit:
```

$$\begin{aligned}&\mathsf{esp} = sp \wedge \mathsf{stack}(sp, ss) * \mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{full}}(cur) * cur-4 \mapsto size * sp \mapsto [size-40-ss]\\&\qquad \wedge ss = sp-cur-40 \wedge ss \geq 8\end{aligned}$$

// *unfold, shuffle*

$$\begin{aligned}&\mathsf{esp} = sp \wedge \mathsf{stack}(sp, ss)\\&\qquad *cur-4 \mapsto size * cur+8 \mapsto [32] * sp \mapsto [size-40-ss] \wedge ss = sp-cur-40 \wedge ss \geq 8\\&\qquad *\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})\\&*\mathsf{mctx\_t}(\mathsf{sched\_env}(\mathsf{I}_{\mathsf{core}}), sched)\end{aligned}$$

```
        mov eax, [mth_cur]
        mov [eax+_state], DEAD   // mth_cur->state = DEAD;
        mov eax, [mctx_sched]
        push eax
```

$$\begin{aligned}&\mathsf{esp} = sp-4 \wedge \mathsf{stack}(sp-4, ss-4, sched)\\&\qquad *cur-4 \mapsto size * cur+8 \mapsto [32] * sp \mapsto [size-40-ss] \wedge ss = sp-cur-40 \wedge ss \geq 8\\&\qquad *\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{DEAD} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})\\&*\mathsf{mctx\_t}(\mathsf{sched\_env}(\mathsf{I}_{\mathsf{core}}), sched)\end{aligned}$$

// *unfold, shuffle, cast*

$$\begin{aligned}&\mathsf{esp} = sp-4 \wedge \mathsf{stack}(sp-4, ss-4, sched)\\&\qquad *cur-4 \mapsto size * cur+8 \mapsto [32] * sp \mapsto [size-40-ss] \wedge ss = sp-cur-40 \wedge ss \geq 8\\&\qquad *\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{DEAD} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})\\&*\mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{DEAD} * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})\\&\qquad *cur-4 \mapsto size * cur+8 \mapsto [size-8], sched)\end{aligned}$$

```
        call loadcontext         // loadcontext(mctx_sched);
```

Figure 7.12: Verification of thread termination

ing invariant. In addition, it combines the TCB and stack of the dead thread into one continuous memory block, which is to be freed by the scheduler.

Threading initialization. Multi-threading is started by calling *mth_start*(). This function spawns the main thread, allocates the scheduler context, and establishes the global data structures used in the threading invariant of Figure 7.8. It starts thread scheduling by invoking *mth_scheduler*(), which will return only after all threads are dead and deallocated. *mth_start*() then deallocates the scheduler context and returns to the user program. We present the verification of *mth_start*() in Figure 7.13.

Fn $stacksize, main, arg$ $\{$ Aux: ; Local : $[72]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto \_ * \mathsf{mth\_cur} \mapsto \_ * \mathsf{mth\_rq} \mapsto \_ * \mathsf{a}_{pre} \wedge stacksize \geq 12$
$\wedge \mathsf{cptr}(main,$ Fn arg' $\{$ Local : $[stacksize - 8]$;
Pre: $arg' = arg \wedge \mathsf{I_{mem}} * \mathsf{I_{mth}} * \mathsf{a}_{pre}$;
Post: $\mathsf{I_{mem}} * \mathsf{I_{mth}}$ $\})$;
Post: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto \_ * \mathsf{mth\_cur} \mapsto \_ * \mathsf{mth\_rq} \mapsto \_ \}$

```
mth_start:                    // void mth_start (int stacksize, void *(*main)(void *), void *arg);
```

Local: $[72]$; Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto \_ * \mathsf{mth\_cur} \mapsto \_ * \mathsf{mth\_rq} \mapsto \_ * \mathsf{a}_{pre} \wedge stacksize \geq 12$
$\wedge \mathsf{cptr}(main,$ Fn arg' $\{$ Local : $[stacksize - 8]$;
Pre: $arg' = arg \wedge \mathsf{I_{mem}} * \mathsf{I_{mth}} * \mathsf{a}_{pre}$;
Post: $\mathsf{I_{mem}} * \mathsf{I_{mth}}$ $\})$;

```
        mov [mth_rq], NULL  // mth_rq = NULL;
```

Local: $[72]$; Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto \_ * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([], \mathsf{mth\_rq}) * \mathsf{a}_{pre}$
$\wedge stacksize \geq 12 \wedge \mathsf{cptr}(main,$ Fn arg' $\{$ Local : $[stacksize - 8]$;
Pre: $arg' = arg \wedge \mathsf{I_{mem}} * \mathsf{I_{mth}} * \mathsf{a}_{pre}$;
Post: $\mathsf{I_{mem}} * \mathsf{I_{mth}}$ $\})$;

```
        mov eax, [esp+12]   // initialize the ready thread queues
        push eax
        mov eax, [esp+8]
        push eax
        mov eax, [esp+4]
        push eax
        call mth_spawn      // mth_spawn(stacksize, main, arg);
        add esp, 12         // spawn a thread for the main program
```

Local: $[72]$; Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto \_ * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([main], \mathsf{mth\_rq})$

```
        push size_of_mctx_st
        call malloc  // mctx_sched = (mctx_t)malloc(sizeof(mctx_st));
        add esp, 4
        mov [mctx_sched], eax
```

Local: $[72]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([main], \mathsf{mth\_rq})$

```
        call mth_scheduler  // mth_scheduler(); // do the threading
```

Local: $[72]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([], \mathsf{mth\_rq})$

```
        mov eax, [mctx_sched]
        push eax
        call free           // free(mctx_sched);
        add esp, 4
```

Local: $[72]$; Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([], \mathsf{mth\_rq})$

```
                            // cast
```

Local: $[72]$; Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto \_ * \mathsf{mth\_cur} \mapsto \_ * \mathsf{mth\_rq} \mapsto \_$

```
        ret
```

Figure 7.13: Verification of thread initialization

Fn { Aux: ; Local: $[16]$;

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$;

Post: $\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([], \mathsf{mth\_rq})$ }

```
mth_scheduler:                    // void mth_scheduler (void);
```

Local: $[16]$;

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$

```
        push mth_rq          // while (mth_cur = queue_delete(&mth_rq)) != NULL)
        call queue_delete
        add esp, 4
```

Local: $[16]$;

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}(tl', \mathsf{mth\_rq})$
$\wedge \mathsf{eax} = retv \wedge ((retv = \mathsf{NULL} \wedge tl = tl' = []) \vee$
$(retv \neq \mathsf{NULL} \wedge tl = retv :: tl' \wedge retv \mapsto \_ * \mathsf{mth\_t}(\mathsf{READY}, \mathsf{I}_{\mathsf{core}}(\mathsf{READY}), retv)))$

```
        cmp eax, NULL
        je sc_ret
```

Local: $[16]$;

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}(tl', \mathsf{mth\_rq})$
$\wedge \mathsf{eax} = retv \wedge retv \neq \mathsf{NULL} \wedge tl = retv :: tl' \wedge retv \mapsto \_ * \mathsf{mth\_t}(\mathsf{READY}, \mathsf{I}_{\mathsf{core}}(\mathsf{READY}), retv)$

```
        mov [mth_cur], eax   // Found next ready thread
        add eax, _mctx
        push eax
        mov eax, [mctx_sched]
        push eax    // ENTERING THREAD - by switching the machine context
```

Local: $[8], sched, retv+8$;

Pre: $\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto retv * \mathsf{ready\_queue}(tl', \mathsf{mth\_rq})$
$\wedge \mathsf{eax} = retv \wedge retv \neq \mathsf{NULL} \wedge tl = retv :: tl' \wedge retv \mapsto \_ * \mathsf{mth\_t}(\mathsf{READY}, \mathsf{I}_{\mathsf{core}}(\mathsf{READY}), retv)$

// *unfold, shuffle, cast, rename*

Local: $[8], sched, retv+8$;

Pre: $sched \mapsto [32]$
$* \mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl', \mathsf{mth\_rq})$
$* \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, \mathsf{READY} * \mathsf{ready\_queue}(tl', \mathsf{mth\_rq})$
$* \mathsf{mctx\_t}(\mathsf{sched\_env}(\mathsf{I}_{\mathsf{core}}), sched), cur+8)$

```
        call swapcontext     // swapcontext(mctx_sched, &mth_cur->mctx);
```

Local: $[8], sched, retv+8$;

Pre: $sched \mapsto [32]$
$* \mathsf{I}_{\mathsf{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * cur \mapsto \_, st * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$* ((st = \mathsf{READY} \wedge \mathsf{mctx\_t}(\mathsf{I}_{\mathsf{mem}} * \mathsf{I}_{\mathsf{core}}(\mathsf{READY}, cur), cur+8)) \vee$
$(st = \mathsf{DEAD} \wedge \exists size.\, cur-4 \mapsto size * cur+8 \mapsto [size-8]))$

```
        add esp, 8
        mov eax, [mth_cur]   // If previous thread is now marked as dead, kick it out
        mov ecx, [eax+_state]
        cmp ecx, DEAD        // if (mth_cur->state == DEAD)
        jne sc_els
```

Figure 7.14: Verification of thread scheduler

Local: $[16]$;
Pre: $sched \mapsto [32] * \mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$\quad \wedge st = \mathsf{DEAD} \wedge cur \mapsto \_, st * cur-4 \mapsto size * cur+8 \mapsto [size-8] \wedge \mathsf{eax} = cur$

```
                          // cast
```

Local: $[16]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto cur * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$\quad \wedge cur-4 \mapsto size * cur \mapsto [size] \wedge \mathsf{eax} = cur$

```
        push eax
        call free         // free(mth_cur); // No stack from now on
        add esp, 4
```

Local: $[16]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto cur * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$

```
        jmp mth_scheduler
```

Local: $[16]$;
Pre: $sched \mapsto [32] * \mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * \mathsf{mth\_cur} \mapsto cur * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$\quad \wedge st = \mathsf{READY} \wedge cur \mapsto \_, st * \mathsf{mctx\_t}(\mathsf{I_{mem}} * \mathsf{I_{core}}(\mathsf{READY}, cur), cur+8) \wedge \mathsf{eax} = cur$

```
sc_els:                  // cast
```

Local: $[16]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto cur * \mathsf{ready\_queue}(tl, \mathsf{mth\_rq})$
$\quad \wedge cur \mapsto \_ * \mathsf{mth\_t}(\mathsf{READY}, \mathsf{I}_{core}(\mathsf{READY}), cur) \wedge \mathsf{eax} = cur$

```
        push eax           // else
        push mth_rq        // insert last running thread back into this queue
        call queue_insert  // queue_insert(&mth_rq, mth_cur);
        add esp, 8
```

Local: $[16]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto cur * \mathsf{ready\_queue}(tl@[cur], \mathsf{mth\_rq})$

```
        jmp mth_scheduler
```

Local: $[16]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}(tl', \mathsf{mth\_rq})$
$\quad \wedge \mathsf{eax} = retv \wedge retv = \mathsf{NULL} \wedge tl = tl' = []$

```
sc_ret:                  // cast
```

Local: $[16]$;
Pre: $\mathsf{I_{mem}} * \mathsf{mctx\_sched} \mapsto sched * sched \mapsto [32] * \mathsf{mth\_cur} \mapsto \_ * \mathsf{ready\_queue}([], \mathsf{mth\_rq})$

```
        ret
```

Figure 7.15: Verification of thread scheduler (continued)

Thread scheduler. *mth_scheduler*() is essentially a loop implementing FIFO scheduling. It is directly invoked only by *mth_start*(). Upon entering *mth_scheduler*(), the scheduler context and some other threading data structures are already in place. If the ready queue is not empty, *mth_scheduler*() removes the first thread, sets it as the current thread, and switches to it using *swapcontext*(). The scheduler resumes execution when the current thread yields or exits, at which point the scheduler either puts the yielding thread into the ready queue or deallocates it based on the thread state. The scheduler finally returns when the ready queue becomes empty. We give the specification of *mth_scheduler*() as follows, again omitting the details for space. The post-condition of *mth_scheduler* preserves everything, except that there no thread in the ready queue.

We present the verification of *mth_scheduler*() in Figure 7.14 and Figure 7.15. The scheduler context to be stored in the threading invariant is captured by *swapcontext*() when the scheduler calls it to switch to the new current thread.

7.4 Discussion

First, we want to emphasize that the threading module refers to other utility modules only through the interfaces. For example, the memory invariant $\mathsf{I}_{\mathsf{mem}}$ is used in the threading specifications, but only propagated abstractly during the verification in the local-reasoning style of separation logic. It is therefore safe to upgrade the implementations of the utility modules without affecting the threading verification. Within the threading module, one may upgrade the FIFO scheduler without affecting other threading routines. In fact, one may even have *mth_start*() be parameterized by a scheduler function with only minor changes in the proof structure.

The large and complex proof naturally raises practicality concerns. However, when being compared with other verification methods and judging practicalities, there are two aspects of our method that need to be taken into consideration.

For comparison with analysis- and test-based software verification methods in general, the important question to ask is what kind of guarantee it can deliver. Many of these tools can automatically find bugs in millions of line of system code, but only in a "best-effort" fashion—both the programmer and the end user expect bugs to appear and patches to be released. In our case, we want to completely exclude certain categories of bugs, as guaranteed by the language-based approach used in this dissertation. So the first choice programmers should make is whether they just want to find bugs in applications or make rigorous claims and provide warranty about kernels.

For comparison with other language-based methods, such as traditional type systems, the important question to ask is what kind of target code is actually supported. Since there is no sound type systems for C and assembly, there have been many efforts on building the complete system kernels in higher-level type safe languages, all with trade-offs in efficiency and/or compatibility, some even with significant changes in programming style. In our case, we require no change to the existing system programming style and code base, thus expect no performance or compatibility issue when putting into real use. In general, we believe that low-level system code, mid-level infrastructure code, and high-level application code require different levels of safety guarantees. Thus their verifications will naturally result in different levels of productivity. In fact, the 6 person-month spent may be a fair price to pay, considering that the code is critical and heavily relied upon.

By targeting at a thread library, a piece of system code that is commonly recognized as complex and never fully verified with traditional approaches, we demonstrate the feasibility of using XCAP frameworks for the mechanical verification of realistic machine-level code. That being said, at the current stage, the abstraction level used in this dissertation is best suited for verifying critical small-scale software such as core system libraries. Much further study is needed to scale the basic framework for better productivity. One direction is to scale the verification from assembly level to C level, which should dramatically increase the productivity of verifications.

Chapter 8

The Coq Implementations

One key feature of the XCAP framework is mechanization. We mechanize not only our machine syntax, machine semantics, assertion languages, interpretations, inference rules, and program specifications and proofs, but also the complete meta theory of XCAP, in a general mathematic logic. Furthermore, we want to direct have the power of higher-order predicate logic, through a shallow embedding. For these reasons, our usage of the Coq proof assistant [58] is to treat it as both a mechanized logic framework and a mechanized meta logic framework, as explained in Section 2.2.

In this chapter, we first discuss our Coq implementations for the meta theory of *PropX* interpretation, for both predicative and impredicative versions. Then we discuss the implementation of XCAP inference rules and meta theory. Based on that, we present the difference with the implementation of XCAP86. Finally, we discuss our implementation for the certified mini thread library in the previous chapter.

We point out that although our implementation and presentation in this dissertation use CiC/Coq as the mechanized meta logic, in theory it is possible to implement XCAP upon other mechanized meta logics.

8.1 Implementation of *PropX*

For the extended propositions in Section 3.2, we define it as the following.

```
Inductive PropX : Type
    := cptr: Word -> (State -> PropX) -> PropX
     | prop: Prop                       -> PropX
     | andx: PropX -> PropX             -> PropX
     | orx : PropX -> PropX             -> PropX
     | impx: PropX -> PropX             -> PropX
     | allx: forall A, (A -> PropX)     -> PropX
     | extx: forall A, (A -> PropX)     -> PropX.
```

The encoding uses higher-order abstract syntax (HOAS) [52] to represent extended predicates. Interpretation of extended propositions is defined as a recursive function.

```
Definition Assertion := State -> PropX.

Definition CdHpSpec := Map Label Assertion.

Fixpoint Itp (P : PropX) (Si : CdHpSpec) {struct P} : Prop
  := match P with
     | cptr l a => lookup Si l a
     | prop p   => p
     | andx P Q => Itp P Si /\ Itp Q Si
     | orx  P Q => Itp P Si \/ Itp Q Si
     | impx P Q => Itp P Si -> Itp Q Si
     | allx A P => forall x, Itp (P x) Si
     | extx A P => exists x, Itp (P x) Si
     end.
```

And the interpretation of assertions is a simple lift.

```
Definition ItP a Si S := Itp (a S) Si.
```

For the XCAP with impredicative polymorphism defined in Section 4.1 and Section 4.3 the HOAS encoding of extended propositions no longer works. The positivity requirement in Coq inductive definition limits the type A of the quantified terms to be of lower level than PropX, which can not be used for impredicative quantifications. We use de Bruijn notations [15] to encode them, but keep using HOAS for all other constructors.

```
Inductive PropX : list Type -> Type :=
  | var : forall L A, A                                 -> PropX (A :: L)
  | lift: forall L A, PropX L                           -> PropX (A :: L)
  | cptr: forall L, Word -> (State -> PropX L)          -> PropX L
  | ref : forall L, Word -> (Word  -> PropX L)          -> PropX L
  | prop: forall L, Prop                                -> PropX L
  | andx: forall L, PropX L -> PropX L                  -> PropX L
  | orx : forall L, PropX L -> PropX L                  -> PropX L
  | impx: forall L, PropX L -> PropX L                  -> PropX L
  | allx: forall L A, (A -> PropX L)                    -> PropX L
  | extx: forall L A, (A -> PropX L)                    -> PropX L
  | allv: forall L A, PropX (L ++ A :: nil)             -> PropX L
  | extv: forall L A, PropX (L ++ A :: nil)             -> PropX L
  | mu  : forall L A, (A -> PropX (L ++ A :: nil)) -> A -> PropX L.
```

We can also define some syntactic sugars for *PropX*.

```
Notation "<< p >>" := (prop _ p  ) (at level 40).
Notation "P ./\ Q" := (andx _ P Q) (at level 51, right associativity).
Notation "P .\/ Q" := (orx  _ P Q) (at level 70, right associativity).
Notation "P .-> Q" := (impx _ P Q) (at level 100, right associativity).
```

The following auxiliary definitions are straight-forward.

```
Definition Assertion := State -> PropX nil.

Definition WordTy    := Word  -> PropX nil.

Definition CdHpSpec  := Map Label Assertion.

Definition DtHpSpec  := Map Label WordTy.

Definition Env       := list (PropX nil).
```

When weak update reference is supported, the interpretation of XCAP assertions can be defined as the following.

```
Definition Itp P Si Fi := OK nil Si Fi P.

Definition DH Si Fi H  :=
  forall l t, lookup Fi l t -> exists w, lookup H l w /\ Itp (t w) Si Fi.

Definition ItP a Si S  :=
  match S with (H, R) =>
    exists Fi, exists H1, exists H2,
      disjoint H1 H2 /\ merge H1 H2 = H /\ Itp (a (H1,R)) Si Fi /\ DH Si Fi H2
  end.

Notation "p ==> q"     := (forall Si S, ItP p Si S -> ItP q Si S)
                          (at level 130, right associativity).
```

Itp is the interpretation of extended propositions, defined using the *PropX* validity below. ItP is the assertion interpretation, as discussed in Section 5.1.

```
Inductive OK : Env -> CdHpSpec -> DtHpSpec -> PropX nil -> Prop :=
  | ok_env    : forall E Si Fi p,   In p E          -> OK E Si Fi p
  | ok_cptr_i : forall E Si Fi l P, lookup Si l P   -> OK E Si Fi (cptr nil l P)
  | ok_cptr_e : forall E Si Fi l P q,
                  OK E Si Fi (cptr nil l P) ->
                  (lookup Si l P -> OK E Si Fi q)   -> OK E Si Fi q
  | ok_ref_i : forall E Si Fi l P, lookup Fi l P    -> OK E Si Fi (ref nil l P)
  | ok_ref_e : forall E Si Fi l P q,
                  OK E Si Fi (ref nil l P) ->
                  (lookup Fi l P -> OK E Si Fi q)   -> OK E Si Fi q
  | ok_prop_i : forall E Si Fi (p : Prop), p        -> OK E Si Fi (<< p >>)
  | ok_prop_e : forall E Si Fi (p : Prop) q,
                  OK E Si Fi (<< p >>) ->
                  (p -> OK E Si Fi q)               -> OK E Si Fi q
  | ok_and_i  : forall E Si Fi p q, OK E Si Fi p ->
                                    OK E Si Fi q    -> OK E Si Fi (p ./\ q)
  | ok_and_e1 : forall E Si Fi p q,
                  OK E Si Fi (p ./\ q)              -> OK E Si Fi p
  | ok_and_e2 : forall E Si Fi p q,
                  OK E Si Fi (p ./\ q)              -> OK E Si Fi q
  | ok_or_i1  : forall E Si Fi p q, OK E Si Fi p    -> OK E Si Fi (p .\/ q)
  | ok_or_i2  : forall E Si Fi p q, OK E Si Fi q    -> OK E Si Fi (p .\/ q)
  | ok_or_e   : forall E Si Fi p q r,
                  OK E Si Fi (p .\/ q) ->
                  OK (cons p E) Si Fi r ->
                  OK (cons q E) Si Fi r             -> OK E Si Fi r
  | ok_imp_i  : forall E Si Fi p q,
                  OK (cons p E) Si Fi q             -> OK E Si Fi (p .-> q)
  | ok_imp_e  : forall E Si Fi p q,
                  OK E Si Fi p ->
                  OK E Si Fi (p .-> q)              -> OK E Si Fi q
  | ok_allx_i : forall E Si Fi A
                  (P : A -> PropX nil),
                  (forall x, OK E Si Fi (P x))      -> OK E Si Fi (allx nil A P)
  | ok_allx_e : forall E Si Fi A P,
                  OK E Si Fi (allx nil A P) ->
                  forall B : A,                        OK E Si Fi (P B)
  | ok_extx_i : forall E Si Fi A
                  (P : A -> PropX nil) x,
                  OK E Si Fi (P x)                  -> OK E Si Fi (extx nil A P)
  | ok_extx_e : forall E Si Fi A P q,
                  OK E Si Fi (extx nil A P) ->
                  (forall B : A,
                    OK (cons (P B) E) Si Fi q)      -> OK E Si Fi q
  | ok_allv_i : forall E Si Fi A p,
                  (forall x : A -> PropX nil,
                    OK E Si Fi (Subst nil A p x))   -> OK E Si Fi (allv nil A p)
  | ok_extv_i : forall E Si Fi A p
                  (q: A -> PropX nil),
                  OK E Si Fi (Subst nil A p q)      -> OK E Si Fi (extv nil A p)
  | ok_mu_i   : forall E Si Fi A p (v:A),
                  OK E Si Fi (Subst nil A (p v)
                                    (mu nil A p))   -> OK E Si Fi (mu nil A p v).
```

Soundness of *PropX* validity rules is implemented as the following lemma. It takes around 4,000 lines of Coq tactics to prove this theorem.

```
Lemma Validity_Soundness :
  forall L Si Fi (p : PropX L),
    match L as t return forall p : PropX t, Prop with
      nil => fun p => OK nil Si Fi p ->
        match p in PropX t return t = nil -> Prop with
        | var  _ _ _ => fun pf => False
        | lift _ _ _ => fun pf => False
        | cptr _ l P => fun pf => lookup Si l (fun S => eq_rect _ _ (P S) _ pf)
        | ref  _ l P => fun pf => lookup Fi l (fun S => eq_rect _ _ (P S) _ pf)
        | prop _ p   => fun pf => p
        | andx _ p q => fun pf => OK nil Si Fi (eq_rect _ _ p _ pf) /\
                                  OK nil Si Fi (eq_rect _ _ q _ pf)
        | orx  _ p q => fun pf => OK nil Si Fi (eq_rect _ _ p _ pf) \/
                                  OK nil Si Fi (eq_rect _ _ q _ pf)
        | impx _ p q => fun pf => OK nil Si Fi (eq_rect _ _ p _ pf) ->
                                  OK nil Si Fi (eq_rect _ _ q _ pf)
        | allx _ A P => fun pf => forall x, OK nil Si Fi (eq_rect _ _ (P x) _ pf)
        | extx _ A P => fun pf => exists x, OK nil Si Fi (eq_rect _ _ (P x) _ pf)
        | allv _ A p => fun pf => forall x, OK nil Si Fi
            (Subst _ _ (eq_rect _ _ p _ (trans_eq (eq_app_eq _ _ _ pf)
                                                  (sym_eq (nil_app_eq _))))
                       x)
        | extv _ A p => fun pf => exists x, OK nil Si Fi
            (Subst _ _ (eq_rect _ _ p _ (trans_eq (eq_app_eq _ _ _ pf)
                                                  (sym_eq (nil_app_eq _))))
                       x)
        | mu _ _ P v => fun pf => OK nil Si Fi
            (Subst _ _ (eq_rect _ _ (P v) _ (trans_eq (eq_app_eq _ _ _ pf)
                                                      (sym_eq (nil_app_eq _))))
                       (mu nil _ (fun v => (eq_rect _ _ (P v)
                                                      _ (eq_app_eq _ _ _ pf)))))
        end (refl_equal _)
    | _ => fun _ => True
    end p.
```

To allow more *Prop*-like handling of proofs, we build many tactics for *PropX* reasoning. Below are a selection of them.

```
Definition okvalid := Validity_Soundness nil.

Ltac des H     :=
  generalize (okvalid _ _ H); clear H; intro; unfold eq_rect in * |-.

Ltac dest H    :=
  generalize (okvalid _ _ H); clear H; destruct 1; unfold eq_rect in * |-.

Ltac destx H v :=
  generalize (okvalid _ _ H); clear H; destruct 1 as [v]; unfold eq_rect in * |-.

Ltac splitx    := apply ok_and_i.

Ltac leftx     := apply ok_or_i1.

Ltac rightx    := apply ok_or_i2.

Ltac introx    := apply ok_allx_i.

Ltac existsx x := apply ok_extx_i with x.

Ltac introv    := apply ok_allv_i.

Ltac existsv a := apply ok_extv_i with a.

Ltac propx     := apply ok_prop_i.

Ltac cptrx     := apply ok_cptr_i.
```

8.2 Implementation of XCAP

We implemented XCAP inference rules as the following inductive definitions.

```
Inductive WFiseq : CdHpSpec -> Assertion -> InstrSeq -> Prop :=
  | wfiseq : forall Si a c I a',
               (forall Si s, ItP a Si s ->
                  exists s',
                    Next c s s' /\ ItP a' Si s') ->
               WFiseq Si a' I                          -> WFiseq Si a (iseq c I)
  | wfbgti : forall Si a rs w l I a' a'',
               ((fun s => << _R s rs <= w >> ./\ a s) ==> a'') ->
               ((fun s => << _R s rs >  w >> ./\ a s) ==> a' ) ->
               lookup Si l a' ->
               WFiseq Si a'' I                    -> WFiseq Si a (bgti rs w l I)
  | wfjd   : forall Si a l a',
               lookup Si l a' ->
               (a ==> a')                                  -> WFiseq Si a (jd l)
  | wfjmp  : forall Si a r,
               (a ==> fun S => extv _ _
                 (eq_rect _ _ (cptr _ (_R S r) (var _ _)
                                ./\ var _ _ S)
                            _ (nil_app_eq _)))            -> WFiseq Si a (jmp r)
  | wfecp  : forall Si a l a' a'' I,
               WFiseq Si a' I  ->
               lookup Si l a'' ->
               ((fun s => cptr _ l a'' ./\ a s) ==> a')         -> WFiseq Si a I.

Inductive WFcode : CdHpSpec -> CodeHeap -> CdHpSpec -> Prop :=
  | wfcdhp : forall Si Si' C,
               (forall l a, lookup Si' l a ->
                  exists I, lookup C l I /\ WFiseq Si a I)    -> WFcode Si C Si'
  | wflink : forall Si1 Si2 Si'1 Si'2 C1 C2,
               WFcode Si1 C1 Si'1 ->
               WFcode Si2 C2 Si'2 ->
               Map.disjoint C1 C2 ->
               subseteq Si1 (merge Si1 Si2) ->
               subseteq Si2 (merge Si1 Si2)
                       -> WFcode (merge Si1 Si2) (merge C1 C2) (merge Si'1 Si'2).

Definition WFprogram Si a P := match P with (c, (s, i)) =>
                                 WFcode Si c Si /\ ItP a Si s /\ WFiseq Si a i
                               end.
```

Selected soundness theorem and lemmas of XCAP are presented as follows. It takes around 700 Coq tactics to prove these theorem. This confirms that XCAP is a lightweight framework.

```
Lemma InstrSeqWeakening :
  forall Si Si' a a' I,
    WFiseq Si a' I -> subseteq Si Si' -> (a ==> a') -> WFiseq Si' a I.

Lemma CodeHeapTyping :
  forall C Si l a,
    WFcode Si C Si -> lookup Si l a -> exists I, lookup C l I /\ WFiseq Si a I.

Theorem Soundness :
  forall Si a P, WFprogram Si a P ->
    exists a', exists P', STEP P P' /\ WFprogram Si a' P'.
```

8.3 Implementation of XCAP86

XCAP86's machine model, Mini86, is quite different from TM used in XCAP. Its Coq implementation is complicated by new details such as byte-addressed word-aligned memory, fixed machine word size, and machine instruction decoding. For example, the following are the macros to test if a byte or word is stored in the memory. (To keep our machine model close to TM, the memory is modeled as a mapping from aligned addresses to machine words.)

```
Definition byte h l :=
  match (Z_eq_dec (l mod 4) 0) with
  | left _ => match (h l) with
              | Some w => Some (w mod 256)
              | None => None
              end
  | _      => match (Z_eq_dec (l mod 4) 1) with
              | left _ => match (h (l-1)) with
                          | Some w => Some ((w / 256) mod 256)
                          | None => None
                          end
              | _      => match (Z_eq_dec (l mod 4) 2) with
                          | left _ => match (h (l-2)) with
                                      | Some w => Some ((w / 65536) mod 256)
                                      | None => None
                                      end
                          | _      => match (h (l-3)) with
                                      | Some w => Some ((w / 65536) / 256)
                                      | None => None
                                      end
                          end
              end
  end.
```

```
Definition dword h l w := byte h l     = Some (w mod 256)
                       /\ byte h (l+1) = Some (w / 256 mod 256)
                       /\ byte h (l+2) = Some (w / 65536 mod 256)
                       /\ byte h (l+3) = Some (w / 65536 / 256).
```

We carefully designed Mini86 and its implementation so that they stay as close to TM as possible. For example, below are selected implementations details of Mini86, which look similar to TM's in Section 2.2.

```
Inductive Next : Instr -> State -> State -> Prop :=
  | stp_add   : forall r o H R F R' F',
                  Dword (R r + Ro R o) ->
                  uR R R' r (R r + Ro R o) ->
                  CalcF F' (R r + Ro R o) ->
                  Next (add r o) (H, R, F) (H, R', F')
  | stp_mov   : forall r o H R F R',
                  uR R R' r (Ro R o) ->
                  Next (mov r o) (H, R, F) (H, R', F)
  | stp_movrm : forall r d H R F w R',
                  lookup H (Ra R d) w ->
                  uR R R' r w ->
                  Next (movrm r d) (H, R, F) (H, R', F)
  | ... .

Inductive STEP : Program -> Program -> Prop :=
  | stp_iseq : forall H R F H' R' F' pc c npc,
                 Dc H pc c npc ->
                 Next c (H, R, F) (H', R', F') ->
                 STEP ((H, R, F), pc) ((H', R', F'), npc)
  | stp_jmp  : forall o H R F pc npc,
                 Dc H pc (jmp o) npc ->
                 STEP ((H, R, F), pc) ((H, R, F), Ro R o)
  | ... .
```

Other than the difference in machine model, XCAP86 also mainly differs in instruction-level inference rules.

```
Inductive WFiseq : CdHpSpec -> Assertion -> InstrSeq -> Prop :=
  | wfiseq  : forall Si a c I a',
                (a ==> (fun s => Ex s' .
                         <<Next c s s'>> ./\ a' s')) ->
                WFiseq Si a' I                         -> WFiseq Si a (iseq c I)
  | wfjcc   : forall Si a cc f I a' a'',
                ((fun s => <<~(Fcc (_F s) cc)>> ./\ a s) ==> a'') ->
                ((fun s => <<Fcc (_F s) cc>> ./\ a s) ==> a') ->
                lookup Si f a' ->
                WFiseq Si a'' I               -> WFiseq Si a (iseq (jcc cc f) I)
  | wfjmpi  : forall Si a f a',
                lookup Si f a' ->
                (a ==> a')                 -> WFiseq Si a (instr (jmp (word f)))
  | wfjmpr  : forall Si a r,
                (a ==> fun S => extv _ _
                  (eq_rect _ _ (cptr _ (_R S r) (var _ _)
                                 ./\ var _ _ S)
                             _ (nil_app_eq _)))
                                            -> WFiseq Si a (instr (jmp (reg r)))
  | wfcalli : forall Si a f fret a',
                lookup Si f a' ->
                (a ==> (fun s => Ex s'.
                  <<Next (push (word fret)) s s'>>
                    ./\ a' s'))
                                    -> WFiseq Si a (instr' (call (word f)) fret)
  | wfcallr : forall Si a r fret,
                (a ==> (fun S => extv _ _
                  (eq_rect _ _ (cptr _ (_R S r) (var _ _)
                                 ./\ Ex s'. <<Next (push (word fret)) S s'>>
                                               ./\ var _ _ s')
                             _ (nil_app_eq _))))
                                     -> WFiseq Si a (instr' (call (reg r)) fret)
  | wfret   : forall Si a,
                (a ==> (fun S => Ex fret.
                   <<lookup (_H S) (_R S esp) fret>>
                     ./\ extv _ _
                   (eq_rect _ _ (cptr _ fret (var _ _)
                                  ./\ Ex s'. <<Next pop' S s'>>
                                               ./\ var _ _ s')
                              _ (nil_app_eq _))))
                                                      -> WFiseq Si a (instr ret)
  | wfecp   : forall Si a f a' a'' I,
                WFiseq Si a' I ->
                lookup Si f a'' ->
                ((fun s => cptr _ f a'' ./\ a s) ==> a')        -> WFiseq Si a I.
```

The implementation of Mini86 takes about 400 lines of Coq code. The implementation of XCAP86 meta theory takes about 1,000 lines of Coq code.

8.4 Implementation of MTH

Every MTH module is specified and verified separately from its clients. During the certification of MTH, we collected a large number of lemmas, including some on generic separation-logic reasoning and others on common code patterns. Our Coq implementation consists of approximately 34,000 lines of Coq code (about 5,000 lines are common lemmas, 3,000 lines are for the queue module, 8,000 lines are for the machine context module, and 18,000 lines are for the threading module). The full compilation of XCAP86 and MTH implementation in Coq (proof-script checking and proof-binary generation) takes about 4 hours on an Intel Pentium M 1.6Ghz CPU with 2GB memory.

The development of the code took nearly six person-month. There are several reasons for the large proof size. First of all, this is the first time we are doing this kind of realistic proof, so a lot of infrastructural code and experience need to be developed and learned. For example, the first procedure we certified, $swapcontext()$, took one person-month and 5,000 lines of Coq code, even though it consists of only 19 lines of assembly code. Second, the relatively low level of proof reuse caused much redundancy. The third reason is the complexity of x86 machine, where features such as finite integer and byte-addressed word-aligned memory are not support very well in Coq yet. Nevertheless, the biggest reason, we believe, is the complexity of the actual machine code, which is obvious based on the discussions in the previous chapter.

For example, the machine context data type in Section 7.2 is implemented as the following, which is more complex than what its meta presentation looks like.

```
Definition mctx_t L aenv mctx : Heap -> PropX L :=
 fun H =>
  Ex retv, bx, cx, dx, si, di, bp, sp, ret.
    star (ptolist _ mctx (retv::bx::cx::dx::si::di::bp::sp::nil))
   (star (pto _ sp ret)
         (fun H => extv _ _ (eq_rect _ _
            (Lift _ _ (var t0 _ H)
         ./\ codeptr _ ret
               (fun S' => match S' with ((H',R'),F') in
                     reg6 _ bx cx dx si di bp (sp+4) R' ./\ << R' eax = retv>>
                 ./\ star (fun H => Shift _ _
                                      (eq_rect _ _ (aenv H) _ (app_nil_eq _)) _)
                    (star (ptolist _ mctx (retv::bx::cx::dx::si::di::bp::sp::nil))
                    (star (pto _ sp ret)
                          (fun H => Lift _ _ (var t0 _ H))))
                     H'
                end))
                                     _ (nil_app_eq _))))
    H.
```

On the other hand, since we have used a lot of abstraction and composition in the verification and implementation, we are still able to achieve very abstract and modular specifications and reasoning, such as the thread yielding function interface shown below.

```
Definition Amth_yield : Assertion := Fn6 {Aux : ; Local : [12] ;
  Pre:  star (Imem _) (Imth _);
  Post: star (Imem _) (Imth _)}.
```

Chapter 9

Conclusion and Future Work

9.1 Conclusion

The formal establishment of low-level system code safety poses great challenges to existing language-based security methods. In particular, generating proof-carrying code for realistic machine binary requires a verification framework that is both expressive and modular. Previously, neither traditional type systems nor Hoare-logic systems can achieve both of these goals without comprise. Type systems in general lacks support of general logic predicate, while ordinary logic assertions can not modularly talk about higher-order programming concepts such as embedded code pointers.

In this dissertation, we propose a hybrid approach to achieve modular and expressive machine code verification. By combining the general logic and semantic subsumptions with syntactic type-like constructs and inference rules, our new framework, XCAP, can be used to write program specification in arbitrary logical predicates, in which higher-order features such as embedded code pointers, impredicative polymorphisms, recursive specifications, and weak update references can be expressed as primitive propositions. Thus, XCAP achieves the expressive power of logic-based approaches and the modularity of type-based approaches.

XCAP is not an isolated result. It can be used as a target of existing certifying compiler, since there is a straight-forward type-preserving translation from a typed assembly language to XCAP. The interaction of TAL and XCAP code is very flexible, as the translation is essentially a shallow embedding of TAL in XCAP's assertion language, *PropX*. For application and system code written in type-safe languages, XCAP is a good platform for these code to interoperate with each other, as well as other certified system kernel module.

The most important goal for XCAP is to support direct verification of realistic system kernel assembly code with the help of an interactive proof assistant. XCAP is ported to XCAP86, with a realistic machine model following the x86 architecture. We show how to use XCAP86 to certify a mini thread library, which could serve as a basis for realistic certified system kernel. Such kind of code is not certifiable with traditional type safe languages, thus our verification is the first of this kind.

One key aspect of the XCAP framework is mechanization. Not only are the machine model, specification languages, and proof of assembly programs fully mechanized in the Coq proof assistant, but the entire meta theory of XCAP and its various extensions are fully implemented in Coq as well. In the end, the only things that need to be trusted by the programmers and users are merely the machine model, mathematical logic, and a tiny proof-checker.

In summary, XCAP is a simple, general, expressive, and modular framework for verification of realistic machine code found in both application and system programs.

9.2 Trusted Computing Base

Given the mixed presentation of the theoretical framework and mechanized implementation in the previous chapters, it may not be obvious what exactly are trusted and untrusted for XCAP and its applications. In this section we discuss the trusted computing base (TCB) from the point of view of a programmer.

Trusted: meta logic. To do any formal reasoning, the programmer has to choose a (mechanized) meta logic, and trust it with respect to his intuition, *i.e.*, agree with the representations and reasoning in it. In our case, the variant of Calculus of Inductive Constructions (CiC), *a.k.a.*, higher-order predicate logic with inductive definitions, is the mechanized meta logic theory that needs to be trusted. The programmer should treat this logic as a consistent one. Yet its consistency is not provable inside itself. In practice, it is usually not a big concern, as the meta theory of the logic has been published and can be examined by any logicians.

Trusted: machine model. First of all, the programmer has to trust the mathematical modeling of the actual hardware, namely, the CPU and the memory. The machine representation on paper and encoding in Coq are merely symbols, while the actual computation is a physical process. Without including formal hardware verification results, it is impossible to prove the correspondence between the symbols and the hardware state. In practice, it is usually not a concern though, as the hardware specifications are publically available and considered correct in general.

Trusted: proof checker. In terms of mechanization, the programmer needs to have a proof checker to mechanically check the validity of proofs with respect to specifications. The proof checker has to be trusted unless it is verified again using some other formal method (which again will have the question of TCB). Note that the proof checker is not equal to the entire Coq proof assistant; only a small part of Coq does the job of proof checking. The majority of Coq source code, such as those dealing with proof searching, do not need to be trusted. This is similar to the "certifying compiler" concept. Although we do not have a stand-alone proof checker for CiC, building one should not take more than a few thousand lines of code. One easy way to raise assurance of the proof checker is to independently develop multiple checkers. This way, even if each checker has bugs that occur 20% of the time, the chance of a faulty proof passing three checkers is less than 1%.

Untrusted: PropX and XCAP theory. Since the entire PropX and XCAP meta theory have been mechanized and proved consistent inside Coq, they do not need to be trusted at all. As shown by the various expansions done in this dissertation, as long as the expanded framework is proved sound in Coq, these extensions are safe.

Trusted: program specifications and safety polices. Whether the program specifications and safety polices are trusted or not is perhaps the most confusing question. The programmer has to realize that, in the end, he has to trust that the program specifications do reflect what is expected of the program's behavior. However, for traditional type systems, one often has to ask "what kind of properties are actually supported". This is because there is a gap between program specifications—types—and the meta logical safety polices. In other words, what a programmer sees (types) is not exactly what he gets (safety policy). He has to trust two things: (1) the program specification reflects the safety policy; and (2) the safety policy is what he wants.

For XCAP, however, we support WYSWYG (What You See is What You Get), as the program specifications are written in logic formulas. The code heap specification contains the safety polices at each key program point. (Strictly speaking, XCAP specifications are not meta logic formulas, as they may contain PropX syntactic constructs such as cptr; however, the soundness theorem of PropX interpretation guarantees that PropX level constructors and quantifiers can be treated as meta logic ones.) Thus it is meaningless to ask "what kind of properties are actually supported", as all possible state-based safety policies definable in CiC are supported in XCAP. When we refer to an XCAP program specification as "trusted", we only mean that we believe the meta logic safety policy (same as the program specification) is what we really want.

Untrusted: safety proof. Once the program specification is set, the safety proof can be constructed through many different ways, such as theorem provers, interactive proof assistants, and certifying compilers. None of these details matters, though, as the programmer does not need to trust these proof—either they are correct and accepted by the checker, or they are incorrect and rejected by the checker.

Based on the above discussions, it is clear that the trusted computing base for XCAP is really small. In the author's opinion, the only further reducible part of the TCB is the proof checker, which can always be replaced by smaller and smarter implementations, so that they can be verified by hand.

9.3 Comparison with the Indexed Approach

One line of work closely related to the approach in this dissertation is the "index-based semantic model" approach [6, 18, 3, 8, 9, 2, 56, 10]. In this section we refer to that approach as the indexed approach.

The indexed approach and XCAP share a lot of similarities as both of them belong to the foundational proof-carrying code architecture. Both are based on mechanized meta logics. While XCAP uses higher-order predicate logic with inductive definitions, the indexed approach has used the same higher-order predicate logic as well as another higher-order logic. Both want to produce foundational proof for well-typed programs and face the same problem of how to modularly express higher-order type-oriented features such as embedded code pointers, general references, impredicative polymorphisms, and recursive types in the meta logic.

The theoretical difference between the indexed approach and the XCAP approach is mainly on whether to use an extra natural number as "index" in "world" and judgments, or to use an extra thin layer of syntax and interpretation to solve the above problem. In the most recent version of the indexed approach, Appel *et al* [10] proposed a new modal model for expressing recursive and impredicative quantified types with mutable reference. Their method uses a Kripke semantics of the Godel-Lob logic of provability. To support mutable reference, they include the memory mapping information in the "world".

At the top level, the difference seems to be rather big considering the similarity between the two approaches. There are several reasons for the different solutions the two approaches have reached as of today.

For the indexed approach, they initially chose to work upon a logic framework (LF), formalize the meta logic (HOL) in it, and build semantic models for types in HOL. This approach turned out to be quite heavyweight, as one has to first build an entire mechanized meta logic through deep embedding. Even that, since there is no built-in inductive definition in the meta logic, one has to use Gödel numbers to simulate simple inductive definitions. While for the CAP/XCAP approach, the work has been based on CiC / Coq, which act as both a logic framework and a meta logic framework (through shallow embedding), and have built-in support of inductive definitions. This allows light-weight implementation, as shown by the hundreds of lines of code for CAP and the thousands of lines of code for PropX and XCAP. Later on, the indexed approach has switched to use CiC/Coq and can now offload the burden of building meta logics and simulating inductive definitions. Since this switch is relatively new, it is still too early to make detailed comparison between the implementations of the latest indexed approach and XCAP.

One major reason is the different guarantee of these two approaches. For XCAP, it is designed to support more than ”type safety” at each program point. The code heap specification specifies the safety policy in general logic predicates (safety policy), and is completely customizable. Meta theory of XCAP guarantees that these safety policies are actually observable. While for the indexed approach, the definition of “codeptr” has always been based on a non-stuckness safety (when the program jumps to an indirect address, the meta theory can only guarantee that the future execution of the program will never get stuck). Thus, there the safety guarantee is more like the traditional notion of type safety and memory safety. It remains unclear how to change the definition of “codeptr” in order to support customizable and observable safety policies without rebuilding the meta theory of the indexed approach.

Another reason is the difference between the verification targets of these approaches. XCAP is designed to support direct verification of assembly programs with non-trivial properties not expressible in traditional types. Besides the examples discussed in this dissertation, various examples of CAP/XCAP programs have been mechanically certified [62, 64, 20, 21, 19]. While the indexed approach has been focusing on type-preserving compilations from high-level languages, thus having few applications of direct verification at assembly level now. It important to note that the XCAP approach also fits into the type-preserving compilation process, as shown by the previous chapters.

We believe that both the indexed approach and the XCAP approach have a long way to go before reaching a fully practical FPCC framework. There are recent trends which show the possibility of these two approaches meeting and merging in the future development.

9.4 Future Work

Given the comprehensive nature of the XCAP framework, there can be many possible future work on type theory, logic theory, verification framework and tools, certifying compilers, safe languages, certified software components, and software security.

General support of higher-order logic specifications One observation from the hybrid type/logic approach being used in XCAP is that, although it currently requires two different forms of syntactic higher-order specifications, for code pointer and data pointer, respectively, there is great similarity in their formation, interpretation, and usage. One idea is to investigate if there is any profound relationship between them, and hopefully to find out a more general way to describe higher-order logic specifications. The intuition is to build a *PropX* level facility whose role and functionality is similar to the inductive definition facility in *Prop*. However, that will raise the question of what exactly the additional layer of syntax does, which we do not have a crystal clear understanding yet. The result here could be very useful for specification and verification of synchronization primitives and garbage collection.

Integrating logic specifications into type systems A different angle to view the framework is to keep existing type systems and add logic-based specifications into them for the additional expressive power. There are already some recent works along this line. However, the goal here is not to define one or a few particular logic-enriched type systems, but to develop a systematic method to inject (partial) logic-power into existing type systems. Hopefully, the effort that is needed to enhance existing typed codes safety guarantee will then be much less.

Infrastructure support for logic-based verification This part includes the selection, evaluation, and potentially development of mathematical logic, binary format, concrete syntax, proof checker, theorem prover, proof assistant, as well as the optimization and delivery of proof. Another important long-term effort will be on building up proof libraries and macros that can be reused. This includes machine architecture, mathematic theory, type theory, logic theory, proof-searching tools, proof-abstraction tools, etc.

Compare various works on type/logic theory The integration of type and logic power into a formal system has been an active area in the past few years. Other than the works on CAP/XCAP, there are works such as Princetons indexed model for types, Harvards hybrid type/logic system, etc. Investigating these works and conducting an comprehensive comparison of them can be useful and might yield interesting results.

General memory management Memory management has been active field in the past decades. In type system world, there are regions, linear types, stack types, *etc.*; while in the logic system world, there are separation logic, BI logic, state logic, CAP/XCAP, etc. Apart from the specific research on certifying garbage collection, it will be interesting to explore the more general memory management models in the logic-based context. For example, it might be eventually possible to let users mix the usage of "managed" and "unmanaged" data pointers and even do intensional analysis of the kind of data pointers.

C-level logic-based language To write system components in a safe language, assembly level is apparently too low, while high level languages might be a bit too high for some cases, especially for legacy systems. The major technical gap between assembly and C lies in: 1) C stack abstraction; 2) register allocation; 3) specification of complex control flow; and 4) preservation after compiler optimization. As the first step, 4) could be ignored or naively handled, since it belongs to the more general question of the interaction between logic-based specification and type-based compilation.

Dynamic code generation / loading / linking As a common runtime feature, dynamic code generation, loading, and linking are difficult to establish their safety, which are crucial for extensible OS and other software. Dynamic linking for type-safe code has been relatively clear by now. However, with components such as runtime being certified using logic-based approach, how safe linking can be automatically and dynamically carried out is still unclear. The most extreme case will be self-modifying code.

Binary compatibility for certified and legacy code One possible and interesting question to ask is whether both certified and legacy code can be compatible with each other. There are two scenarios. The first is to have certified component being deployed in a legacy system, knowing that internally the component should not contain any bug, to increase the overall dependability of systems, as well as to help debugging other legacy components. This is relatively easy to do. The second scenario is to have certified runtime create protected virtual machine to load and run uncertified legacy code, and enforce strict check upon its interface, thus achieving a safe and legacy-friendly system. Communications between certified and legacy code in such systems will be interesting, too.

BIOS / firmware / boot loader No matter how the OS and software stack are verified, the layer between software and the actual "safe" hardware, however thin it is, is still a major venerable part of the entire system. Given the ability to update firmware in most of the modern computing devices, it is difficult to maintain the system integrity. Very likely, emerging security features such as TPM and DRM will also require safety guarantees of firmware to prevent them from being compromised by all possible software-based attacks. These code are suitable for the low-level logic-based verification. Also, there are many interesting non-safety properties for these code. For example, for a firmware, one of the most important features is that it will allow future update what so ever. It is interesting to certify this particular property in the case of power loss during an update.

Preemptive threading (interrupts, I/O, etc.) In the work on verifying threading in this dissertation, as there are no interrupts, only co-operative non-preemptive scheduling is supported. One argument is that preemptive scheduling can be treated as a special case of non-preemptive threading, by inserting a virtual yield instruction between every instruction that has the interrupt bit enabled. It will be interesting to see what kind of re-structuring is necessary to make the specification and proof reasonably simple. I/O will be then meaningful to support with the presence of hardware interrupts.

Device driver Following up work will naturally be device drivers, which currently are either unsafe (Windows) or restricted (Singularity). The vision is that, even for device drivers as complex as self-modifying ones, they can indeed be certified without any compromise. Properties such as temporal ones might need to be introduced here if correctness is a concern. Domain-specific type/logic languages might be defined to utilize the commonality of device drivers. Partial correctness and simple liveness properties might need insights from existing work on device-driver verification, such as model checking.

Garbage collectors It is expected that verification of partial-correctness of a garbage collection process requires more than the plain type safety provided by typical typed languages. Under the hood, one major problem for verification of garbage collection is that traditionally we do not know how to flexibly describe and cast between both strong and weak update reference cell, as well as any reference cell with mixed properties between strong and weak update, *e.g.*, a reference cell with at most n alias.

High-level logic-based language It may be desirable to write the majority of the system kernel and service in high-level languages. It will be interesting to discover how to 1) increase the expressive power of specification languages so more advanced code and properties can be included; 2) increase the level of assurance by using sound logic and checker; 3) integrate the work with other levels of logic-based verification, such as those used by runtime.

Synchronization primitives (mutex, channel, etc.) In the work on verifying threading in this dissertation, only basic services such as creation, yielding, and termination of threads are supported. However, in practice, the most interesting feature of thread libraries for concurrent programming is the set of synchronization primitives it provides. Which set of primitives to support and how they are implemented depends heavily on the concurrent programming model, machine mode, requirement on throughput and response time, as well as the external module interface to be used together with the library. The work will not only yield a set of certified implementations of synchronization primitives, but also track down the precise interfaces of them, which will be interesting by themselves. It might also be connected to existing works on different synchronization and communication methods, such as TLA, allowing us to do an exact comparison of different concurrency models and potentially make code that is based on different concurrency models work together.

Security properties As an important feature of system APIs, security features and the properties they guarantee have traditionally been enforced using either dynamic machine semantics or static semantics of security types. With the runtime and other kernel service being certified with logic specifications, it is possible to use logic formulas to describe security policies, thus allowing security guarantees that are much more flexible.

Logic specifications in certifying compiler Even if we have a higher-level language with logic specifications, right now it is unclear how it would fit into the certifying compiler. Once decidable type checking and typing derivation are replaced by logic proofs, the optimization process will lose the crucial structural information it need to transform the proof. On the other hand, the ability to use arbitrary logic specifications during the compilation process might eventually provide more expressive power on the kinds of optimizations supported comparing with the traditional type-preserving compilation approach, thus producing better optimized certifying compiler.

Appendix A

PropX Validity Soundness Proof

In this section we give the proof structure for the soundness of *PropX* validity rules, as introduced in Section 4.1, Section 4.3, Section 5.1, and Section 6.2. To avoid confusions, we first present the complete *PropX* definition in Figure A.1 and the complete set of *PropX* validity rules in Figure A.2.

We present the soundness of *PropX* interpretation (validity rules), as previously discussed in Theorem 4.1 and Theorem 6.1, as follows.

Theorem A.1 (Soundness of XCAP *PropX* Interpretation)

1. If $[\![\, \langle p \rangle \,]\!]_{\Psi,\Phi}$ then p;
2. if $[\![\, \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \,]\!]_{\Psi,\Phi}$ then $\Psi(\mathtt{f}) = \mathtt{a}$;
3. if $[\![\, \mathsf{ref}(\mathtt{l}, \mathtt{t}) \,]\!]_{\Psi,\Phi}$ then $\Phi(\mathtt{l}) = \mathtt{t}$;
4. if $[\![\mathtt{P} \Wedge \mathtt{Q}]\!]_{\Psi,\Phi}$ then $[\![\mathtt{P}]\!]_{\Psi,\Phi}$ and $[\![\mathtt{Q}]\!]_{\Psi,\Phi}$;
5. if $[\![\mathtt{P} \Vee \mathtt{Q}]\!]_{\Psi,\Phi}$ then either $[\![\mathtt{P}]\!]_{\Psi,\Phi}$ or $[\![\mathtt{Q}]\!]_{\Psi,\Phi}$;
6. if $[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_{\Psi,\Phi}$ and $[\![\mathtt{P}]\!]_{\Psi,\Phi}$ then $[\![\mathtt{Q}]\!]_{\Psi,\Phi}$;
7. if $[\![\forall x{:}A.\,\mathtt{P}]\!]_{\Psi,\Phi}$ and $B{:}A$ then $[\![\,\mathtt{P}[B/x]\,]\!]_{\Psi,\Phi}$;
8. if $[\![\, \exists x{:}A.\,\mathtt{P}]\!]_{\Psi,\Phi}$ then there exists $B{:}A$ such that $[\![\,\mathtt{P}[B/x]\,]\!]_{\Psi,\Phi}$;

$$
\begin{array}{rlll}
(\mathit{PropX}) & \mathtt{P},\mathtt{Q} & ::= \langle p \rangle & \textit{lifted meta proposition} \\
 & & \mid \mathsf{cptr}(\mathtt{f},\mathtt{a}) & \textit{embedded code pointer} \\
 & & \mid \mathsf{ref}(\mathtt{l},\mathtt{t}) & \textit{referencecellpointer} \\
 & & \mid \mathtt{P} \mathbin{\wedge\!\!\wedge} \mathtt{Q} & \textit{conjunction} \\
 & & \mid \mathtt{P} \mathbin{\vee\!\!\vee} \mathtt{Q} & \textit{disjunction} \\
 & & \mid \mathtt{P} \twoheadrightarrow \mathtt{Q} & \textit{implication} \\
 & & \mid \forall x{:}A.\,\mathtt{P} & \textit{universal quantification} \\
 & & \mid \exists x{:}A.\,\mathtt{P} & \textit{existential quantification} \\
 & & \mid \forall \alpha{:}A \to \mathit{PropX}.\,\mathtt{P} & \textit{impredicative universal quantification} \\
 & & \mid \exists \alpha{:}A \to \mathit{PropX}.\,\mathtt{P} & \textit{impredicative existential quantification} \\
 & & \mid (\boldsymbol{\mu}\,\alpha{:}A \to \mathit{PropX}.\,\lambda x{:}A.\,\mathtt{P}\;\; B) & \textit{recursive specification} \\
(\mathit{CdHpSpec}) & \Psi & ::= \{\mathtt{f} \leadsto \mathtt{a}\}^{*} & \\
(\mathit{DtHpSpec}) & \Phi & ::= \{\mathtt{l} \leadsto \mathtt{t}\}^{*} & \\
(\mathit{Assertion}) & \mathtt{a} & \in \mathit{State} \to \mathit{PropX} & \\
(\mathit{WordTy}) & \mathtt{t} & \in \mathit{Word} \to \mathit{PropX} & \\
(\mathit{Env}) & \Gamma & ::= \cdot \mid \Gamma,\mathtt{P} &
\end{array}
$$

Figure A.1: Assertion language of the full-featured XCAP

$\boxed{\Gamma \vdash_{\Psi,\Phi} \mathtt{P}}$ ***(Validity of Extended Propositions)***

(The following presentation omits the Ψ and Φ in judgment $\Gamma \vdash_{\Psi,\Phi} P$.)

$$\frac{\mathtt{P} \in \Gamma}{\Gamma \vdash \mathtt{P}}\ (\textsc{env}) \qquad \frac{p}{\Gamma \vdash \langle p \rangle}\ (\langle\rangle\text{-}\textsc{i}) \qquad \frac{\Gamma \vdash \langle p \rangle \qquad p \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\ (\langle\rangle\text{-}\textsc{e})$$

$$\frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \vdash \mathsf{cptr}(\mathtt{f},\mathtt{a})}\ (\textsc{cp-i}) \qquad \frac{\Gamma \vdash \mathsf{cptr}(\mathtt{f},\mathtt{a}) \qquad (\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\ (\textsc{cp-e})$$

$$\frac{\Phi(\mathtt{l}) = \mathtt{t}}{\Gamma \vdash \mathsf{ref}(\mathtt{l},\mathtt{t})}\ (\textsc{rf-i}) \qquad \frac{\Gamma \vdash \mathsf{ref}(\mathtt{l},\mathtt{t}) \qquad (\Phi(\mathtt{l}) = \mathtt{t}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash !\mathtt{Q}}\ (\textsc{rf-e})$$

$$\frac{\Gamma \vdash \mathtt{P} \qquad \Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}}\ (\mathbin{\wedge\!\!\!\wedge}\text{-}\textsc{i}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}}{\Gamma \vdash \mathtt{P}}\ (\mathbin{\wedge\!\!\!\wedge}\text{-}\textsc{e}1) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q}}{\Gamma \vdash \mathtt{Q}}\ (\mathbin{\wedge\!\!\!\wedge}\text{-}\textsc{e}2)$$

$$\frac{\Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q}}\ (\mathbin{\vee\!\!\!\vee}\text{-}\textsc{i}1) \qquad \frac{\Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q}}\ (\mathbin{\vee\!\!\!\vee}\text{-}\textsc{i}2) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q} \qquad \Gamma, \mathtt{P} \vdash \mathtt{R} \qquad \Gamma, \mathtt{Q} \vdash \mathtt{R}}{\Gamma \vdash \mathtt{R}}\ (\mathbin{\vee\!\!\!\vee}\text{-}\textsc{e})$$

$$\frac{\Gamma, \mathtt{P} \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q}}\ (\twoheadrightarrow\text{-}\textsc{i}) \qquad \frac{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \qquad \Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{Q}}\ (\twoheadrightarrow\text{-}\textsc{e})$$

$$\frac{\Gamma \vdash \mathtt{P}[B/x] \qquad \forall\, B : A}{\Gamma \vdash \forall\!\!\!\forall x : A.\, \mathtt{P}}\ (\forall\!\!\!\forall\text{-}\textsc{i}1) \qquad \frac{\Gamma \vdash \forall\!\!\!\forall x : A.\, \mathtt{P} \qquad B : A}{\Gamma \vdash \mathtt{P}[B/x]}\ (\forall\!\!\!\forall\text{-}\textsc{e}1)$$

$$\frac{B : A \qquad \Gamma \vdash \mathtt{P}[B/x]}{\Gamma \vdash \exists\!\!\!\exists x : A.\, \mathtt{P}}\ (\exists\!\!\!\exists\text{-}\textsc{i}1) \qquad \frac{\Gamma \vdash \exists\!\!\!\exists x : A.\, \mathtt{P} \qquad \Gamma, \mathtt{P}[B/x] \vdash \mathtt{Q} \qquad \forall\, B : A}{\Gamma \vdash \mathtt{Q}}\ (\exists\!\!\!\exists\text{-}\textsc{e}1)$$

$$\frac{\Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \qquad \forall\, \mathtt{a} : A \to \mathit{PropX}}{\Gamma \vdash \forall\!\!\!\forall \alpha : A \to \mathit{PropX}.\, \mathtt{P}}\ (\forall\!\!\!\forall\text{-}\textsc{i}2) \qquad \frac{\mathtt{a} : A \to \mathit{PropX} \qquad \Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha]}{\Gamma \vdash \exists\!\!\!\exists \alpha : A \to \mathit{PropX}.\, \mathtt{P}}\ (\exists\!\!\!\exists\text{-}\textsc{i}2)$$

$$\frac{B : A \qquad \Gamma \vdash \mathtt{P}[B/x][\boldsymbol{\mu}\alpha : A \to \mathit{PropX}.\, \lambda x : A.\, \mathtt{P}/\alpha]}{\Gamma \vdash (\boldsymbol{\mu}\, \alpha : A \to \mathit{PropX}.\, \lambda x : A.\, \mathtt{P}\ \ B)}\ (\mu\text{-}\textsc{i})$$

Figure A.2: Validity rules for extended propositions of the full-featured XCAP

9. if $[\![\forall\alpha{:}A\to PropX.\,\mathtt{P}]\!]_{\Psi,\Phi}$ and $\mathtt{a}{:}A\to PropX$ then $[\![\mathtt{P}[\mathtt{a}/\alpha]\,]\!]_{\Psi,\Phi}$;

10. if $[\![\exists\alpha{:}A\to PropX.\,\mathtt{P}]\!]_{\Psi,\Phi}$ then there exists $\mathtt{a}{:}A\to PropX$ such that $[\![\mathtt{P}[\mathtt{a}/\alpha]\,]\!]_{\Psi,\Phi}$;

11. if $[\![(\boldsymbol{\mu}\,\alpha.\,\lambda x{:}A.\,\mathtt{P}\;\;B)]\!]_{\Psi,\Phi}$ then $[\![\mathtt{P}[B/x][(\boldsymbol{\mu}\,\alpha.\,\lambda x{:}A.\,\mathtt{P}\;\;)/\alpha]\,]\!]_{\Psi,\Phi}$.

Corollary A.2 (XCAP Consistency) $[\![\langle\mathsf{False}\rangle]\!]_{\Psi,\Phi}$ is not provable.

For the proof, we follow the syntactic normalization proof methods in Pfenning [51].

The validity of extended propositions rules of form $\Gamma\vdash_{\Psi,\Phi}\mathtt{P}$ in Figure A.2 are natural deduction rules. We classify them into the following two kinds (and call them altogether as **normal natural validity rules**) as shown in Figure A.3.

$\Gamma\vdash_{\Psi,\Phi}\mathtt{P}\Uparrow$ Extended Proposition P has a normal deduction, and

$\Gamma\vdash_{\Psi,\Phi}\mathtt{P}\downarrow$ Extended Proposition P is extracted from a hypodissertation.

We define the **annotated natural deduction rules** by annotating each normal validity rules with a "+" symbol as $\Gamma\vdash^{+}_{\Psi,\Phi}\mathtt{P}\Uparrow$ and $\Gamma\vdash^{+}_{\Psi,\Phi}\mathtt{P}\downarrow$ and adding the following coercion rule.

$$\frac{\Gamma\vdash^{+}_{\Psi,\Phi}\mathtt{P}\Uparrow}{\Gamma\vdash^{+}_{\Psi,\Phi}\mathtt{P}\downarrow}\;(\textsc{coer}')$$

We then define the **sequent style validity rules** of form $\Gamma\Longrightarrow_{\Psi,\Phi}\mathtt{P}$ in Figure A.4 and extend the sequent rules by annotating sequent judgments with a "+" as $\Gamma\Longrightarrow^{+}_{\Psi,\Phi}\mathtt{P}$ and adding the cut rule.

$$\frac{\Gamma\Longrightarrow^{+}_{\Psi,\Phi}\mathtt{P}\qquad\Gamma,\mathtt{P}\Longrightarrow^{+}_{\Psi,\Phi}\mathtt{Q}}{\Gamma\Longrightarrow^{+}_{\Psi,\Phi}\mathtt{Q}}\;(\textsc{cut})$$

The normalization proof process is:

$$\Gamma\vdash_{\Psi,\Phi}\mathtt{P}\;\;\xrightarrow{A.5}\;\;\Gamma\vdash^{+}_{\Psi,\Phi}\mathtt{P}\Uparrow\;\;\xrightarrow{A.9}\;\;\Gamma\Longrightarrow^{+}_{\Psi,\Phi}\mathtt{P}\;\;\xrightarrow{A.11}\;\;\Gamma\Longrightarrow_{\Psi,\Phi}\mathtt{P}\;\;\xrightarrow{A.6}\;\;\Gamma\vdash_{\Psi,\Phi}\mathtt{P}\Uparrow.$$

First, natural deduction derivations are mapped to derivations in sequent validity rules with cut. Then we do cut-elimination in sequent rules, and map the new cut-free sequent derivation back to a normal natural deduction derivation. Soundness of *PropX* interpretation can then be proved, since the last rule in a normal natural deduction derivation must be one of the introduction rules. We prove the following theorems to construct

$\boxed{\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \Uparrow \qquad \Gamma \vdash_{\Psi,\Phi} \mathtt{P} \downarrow}$ **(*Validity of Extended Propositions*)**

(The following rules omits the Ψ and Φ in judgments $\Gamma \vdash_{\Psi,\Phi} P \Uparrow$ and $\Gamma \vdash_{\Psi,\Phi} P \downarrow$.)

$$\frac{\Gamma \vdash \mathtt{P} \downarrow}{\Gamma \vdash \mathtt{P} \Uparrow}\ (\text{COER}) \qquad \frac{\mathtt{P} \in \Gamma}{\Gamma \vdash \mathtt{P} \downarrow}\ (\text{ENV})$$

$$\frac{p}{\Gamma \vdash \langle p \rangle \Uparrow}\ (\langle\rangle\text{-I}) \qquad \frac{\Gamma \vdash \langle p \rangle \downarrow \qquad p \supset (\Gamma \vdash \mathtt{Q} \Uparrow)}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\langle\rangle\text{-E})$$

$$\frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \vdash \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \Uparrow}\ (\text{CP-I}) \qquad \frac{\Gamma \vdash \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \downarrow \qquad (\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma \vdash \mathtt{Q} \Uparrow)}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\text{CP-E})$$

$$\frac{\Phi(\mathtt{l}) = \mathtt{t}}{\Gamma \vdash \mathsf{ref}(\mathtt{l}, \mathtt{t}) \Uparrow}\ (\text{RF-I}) \qquad \frac{\Gamma \vdash \mathsf{ref}(\mathtt{l}, \mathtt{t}) \downarrow \qquad (\Phi(\mathtt{l}) = \mathtt{t}) \supset (\Gamma \vdash \mathtt{Q} \Uparrow)}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\text{RF-E})$$

$$\frac{\Gamma \vdash \mathtt{P} \Uparrow \qquad \Gamma \vdash \mathtt{Q} \Uparrow}{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q} \Uparrow}\ (\mathbin{\wedge\!\!\!\wedge}\text{-I}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q} \downarrow}{\Gamma \vdash \mathtt{P} \downarrow}\ (\mathbin{\wedge\!\!\!\wedge}\text{-E1}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\wedge\!\!\!\wedge} \mathtt{Q} \downarrow}{\Gamma \vdash \mathtt{Q} \downarrow}\ (\mathbin{\wedge\!\!\!\wedge}\text{-E2})$$

$$\frac{\Gamma \vdash \mathtt{P} \Uparrow}{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q} \Uparrow}\ (\mathbin{\vee\!\!\!\vee}\text{-I1}) \qquad \frac{\Gamma \vdash \mathtt{Q} \Uparrow}{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q} \Uparrow}\ (\mathbin{\vee\!\!\!\vee}\text{-I2}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\vee\!\!\!\vee} \mathtt{Q} \downarrow \qquad \Gamma, \mathtt{P} \vdash \mathtt{R} \Uparrow \qquad \Gamma, \mathtt{Q} \vdash \mathtt{R} \Uparrow}{\Gamma \vdash \mathtt{R} \Uparrow}\ (\mathbin{\vee\!\!\!\vee}\text{-E})$$

$$\frac{\Gamma, \mathtt{P} \vdash \mathtt{Q} \Uparrow}{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \Uparrow}\ (\twoheadrightarrow\text{-I}) \qquad \frac{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \downarrow \qquad \Gamma \vdash \mathtt{P} \Uparrow}{\Gamma \vdash \mathtt{Q} \downarrow}\ (\twoheadrightarrow\text{-E})$$

$$\frac{\Gamma \vdash \mathtt{P}[B/x] \Uparrow \qquad \forall\, B : A}{\Gamma \vdash \forall\!\!\!\forall x : A.\, \mathtt{P} \Uparrow}\ (\forall\!\!\!\forall\text{-I1}) \qquad \frac{\Gamma \vdash \forall\!\!\!\forall x : A.\, \mathtt{P} \downarrow \qquad B : A}{\Gamma \vdash \mathtt{P}[B/x] \downarrow}\ (\forall\!\!\!\forall\text{-E1})$$

$$\frac{B : A \qquad \Gamma \vdash \mathtt{P}[B/x] \Uparrow}{\Gamma \vdash \exists\!\!\!\exists x : A.\, \mathtt{P} \Uparrow}\ (\exists\!\!\!\exists\text{-I1}) \qquad \frac{\Gamma \vdash \exists\!\!\!\exists x : A.\, \mathtt{P} \downarrow \qquad \Gamma, \mathtt{P}[B/x] \vdash \mathtt{Q} \Uparrow \qquad \forall\, B : A}{\Gamma \vdash \mathtt{Q} \Uparrow}\ (\exists\!\!\!\exists\text{-E1})$$

$$\frac{\Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \Uparrow \qquad \forall\, \mathtt{a} : A \to \mathit{PropX}}{\Gamma \vdash \forall\!\!\!\forall \alpha : A \to \mathit{PropX}.\, \mathtt{P} \Uparrow}\ (\forall\!\!\!\forall\text{-I2}) \qquad \frac{\mathtt{a} : A \to \mathit{PropX} \qquad \Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha] \Uparrow}{\Gamma \vdash \exists\!\!\!\exists \alpha : A \to \mathit{PropX}.\, \mathtt{P} \Uparrow}\ (\exists\!\!\!\exists\text{-I2})$$

$$\frac{B : A \qquad \Gamma \vdash \mathtt{P}[B/x][\boldsymbol{\mu}\alpha : A \to \mathit{PropX}.\, \lambda x : A.\, \mathtt{P}/\alpha] \Uparrow}{\Gamma \vdash (\boldsymbol{\mu}\, \alpha : A \to \mathit{PropX}.\, \lambda x : A.\, \mathtt{P}\ \ B) \Uparrow}\ (\mu\text{-I})$$

Figure A.3: Normal natural deduction validity rules

$\boxed{\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{P}}$ ***(Validity of Extended Propositions)***

(The following rules omits the Ψ and Φ in judgment $\Gamma \Longrightarrow_{\Psi,\Phi} P$.)

$$\frac{\mathtt{P} \in \Gamma}{\Gamma \Longrightarrow \mathtt{P}}\ (\textsc{Init}) \qquad \frac{p}{\Gamma \Longrightarrow \langle p \rangle}\ (\langle\rangle\text{-}\textsc{R}) \qquad \frac{p\ \supset\ (\Gamma, \langle p \rangle \Longrightarrow \mathtt{Q})}{\Gamma, \langle p \rangle \Longrightarrow \mathtt{Q}}\ (\langle\rangle\text{-}\textsc{L})$$

$$\frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \Longrightarrow \mathsf{cptr}(\mathtt{f}, \mathtt{a})}\ (\textsc{CP-R}) \qquad \frac{(\Psi(\mathtt{f}) = \mathtt{a})\ \supset\ (\Gamma, \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \Longrightarrow \mathtt{Q})}{\Gamma, \mathsf{cptr}(\mathtt{f}, \mathtt{a}) \Longrightarrow \mathtt{Q}}\ (\textsc{CP-L})$$

$$\frac{\Phi(\mathtt{l}) = \mathtt{t}}{\Gamma \Longrightarrow \mathsf{ref}(\mathtt{l}, \mathtt{t})}\ (\textsc{RF-R}) \qquad \frac{(\Phi(\mathtt{l}) = \mathtt{t})\ \supset\ (\Gamma, \mathsf{ref}(\mathtt{l}, \mathtt{t}) \Longrightarrow \mathtt{Q})}{\Gamma, \mathsf{ref}(\mathtt{l}, \mathtt{t}) \Longrightarrow \mathtt{Q}}\ (\textsc{RF-L})$$

$$\frac{\Gamma \Longrightarrow \mathtt{P} \quad \Gamma \Longrightarrow \mathtt{Q}}{\Gamma \Longrightarrow \mathtt{P} \Wedge \mathtt{Q}}\ (\Wedge\text{-}\textsc{R}) \qquad \frac{\Gamma, \mathtt{P} \Wedge \mathtt{Q}, \mathtt{P} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \Wedge \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\Wedge\text{-}\textsc{L}1) \qquad \frac{\Gamma, \mathtt{P} \Wedge \mathtt{Q}, \mathtt{Q} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \Wedge \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\Wedge\text{-}\textsc{L}2)$$

$$\frac{\Gamma \Longrightarrow \mathtt{P}}{\Gamma \Longrightarrow \mathtt{P} \Vee \mathtt{Q}}\ (\Vee\text{-}\textsc{R}1) \qquad \frac{\Gamma \Longrightarrow \mathtt{Q}}{\Gamma \Longrightarrow \mathtt{P} \Vee \mathtt{Q}}\ (\Vee\text{-}\textsc{R}2) \qquad \frac{\Gamma, \mathtt{P} \Vee \mathtt{Q}, \mathtt{P} \Longrightarrow \mathtt{R} \qquad \Gamma, \mathtt{P} \Vee \mathtt{Q}, \mathtt{Q} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \Vee \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\Vee\text{-}\textsc{L})$$

$$\frac{\Gamma, \mathtt{P} \Longrightarrow \mathtt{Q}}{\Gamma \Longrightarrow \mathtt{P} \twoheadrightarrow \mathtt{Q}}\ (\twoheadrightarrow\text{-}\textsc{R}) \qquad \frac{\Gamma, \mathtt{P} \twoheadrightarrow \mathtt{Q} \Longrightarrow \mathtt{P} \qquad \Gamma, \mathtt{P} \twoheadrightarrow \mathtt{Q}, \mathtt{P} \Longrightarrow \mathtt{R}}{\Gamma, \mathtt{P} \twoheadrightarrow \mathtt{Q} \Longrightarrow \mathtt{R}}\ (\twoheadrightarrow\text{-}\textsc{L})$$

$$\frac{\Gamma \Longrightarrow \mathtt{P}[B/x] \qquad \forall\, B{:}A}{\Gamma \Longrightarrow \mathbb{\forall} x{:}A.\,\mathtt{P}}\ (\mathbb{\forall}\text{-}\textsc{R}1) \qquad \frac{\Gamma, \mathbb{\forall} x{:}A.\,\mathtt{P}, \mathtt{P}[B/x] \Longrightarrow \mathtt{Q} \qquad B{:}A}{\Gamma, \mathbb{\forall} x{:}A.\,\mathtt{P} \Longrightarrow \mathtt{Q}}\ (\mathbb{\forall}\text{-}\textsc{L}1)$$

$$\frac{B{:}A \qquad \Gamma \Longrightarrow \mathtt{P}[B/x]}{\Gamma \Longrightarrow \mathbb{\exists} x{:}A.\,\mathtt{P}}\ (\mathbb{\exists}\text{-}\textsc{R}1) \qquad \frac{\Gamma, \mathbb{\exists} x{:}A.\,\mathtt{P}, \mathtt{P}[B/x] \Longrightarrow \mathtt{Q} \qquad \forall\, B{:}A}{\Gamma, \mathbb{\exists} x{:}A.\,\mathtt{P} \Longrightarrow \mathtt{Q}}\ (\mathbb{\exists}\text{-}\textsc{L}1)$$

$$\frac{\Gamma \Longrightarrow \mathtt{P}[\mathtt{a}/\alpha] \qquad \forall\, \mathtt{a}{:}A \to \mathit{PropX}}{\Gamma \Longrightarrow \mathbb{\forall} \alpha{:}A \to \mathit{PropX}.\,\mathtt{P}}\ (\mathbb{\forall}\text{-}\textsc{R}2) \qquad \frac{\mathtt{a}{:}A \to \mathit{PropX} \qquad \Gamma \Longrightarrow \mathtt{P}[\mathtt{a}/\alpha]}{\Gamma \Longrightarrow \mathbb{\exists} \alpha{:}A \to \mathit{PropX}.\,\mathtt{P}}\ (\mathbb{\exists}\text{-}\textsc{R}2)$$

$$\frac{B{:}A \qquad \Gamma \Longrightarrow \mathtt{P}[B/x][\boldsymbol{\mu}\alpha{:}A \to \mathit{PropX}.\,\lambda x{:}A.\,\mathtt{P}/\alpha]}{\Gamma \Longrightarrow (\boldsymbol{\mu}\,\alpha{:}A \to \mathit{PropX}.\,\lambda x{:}A.\,\mathtt{P}\ \ B)}\ (\mu\text{-}\textsc{R})$$

Figure A.4: Sequent style validity rules

the proof. We use structural induction in most of the proof. The full proof has been mechanized in the Coq proof assistant.

Theorem A.3 (Soundness of Normal Deductions)

1. If $\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \Uparrow$ then $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$, and
2. if $\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \downarrow$ then $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$.

Theorem A.4 (Soundness of Annotated Deductions)

1. If $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \Uparrow$ then $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$, and
2. if $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \downarrow$ then $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$.

Theorem A.5 (Completeness of Annotated Deductions)

1. If $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$ then $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \Uparrow$, and
2. if $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$ then $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \downarrow$.

Theorem A.6 (Soundness of Sequent Calculus)

If $\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{P}$ then $\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \Uparrow$.

Theorem A.7 (Completeness of Sequent Derivations)

1. If $\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \Uparrow$ then $\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{P}$, and
2. if $\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \downarrow$ and $\Gamma, \mathtt{P} \Longrightarrow_{\Psi,\Phi} \mathtt{Q}$ then $\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{Q}$.

Theorem A.8 (Soundness of Sequent Calculus with Cut)

If $\Gamma \Longrightarrow^{+}_{\Psi,\Phi} \mathtt{P}$ then $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \Uparrow$.

Theorem A.9 (Completeness of Sequent Calculus with Cut)

1. If $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \Uparrow$ then $\Gamma \Longrightarrow^{+}_{\Psi,\Phi} \mathtt{P}$, and
2. if $\Gamma \vdash^{+}_{\Psi,\Phi} \mathtt{P} \downarrow$ and $\Gamma, \mathtt{P} \Longrightarrow^{+}_{\Psi,\Phi} \mathtt{Q}$ then $\Gamma \Longrightarrow^{+}_{\Psi,\Phi} \mathtt{Q}$.

Theorem A.10 (Admissibility of Cut)

If $\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{P}$ and $\Gamma, \mathtt{P} \Longrightarrow_{\Psi,\Phi} \mathtt{Q}$ then $\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{Q}$.

Theorem A.11 (Cut Elimination)

If $\Gamma \Longrightarrow^{+}_{\Psi,\Phi} \mathtt{P}$ then $\Gamma \Longrightarrow_{\Psi,\Phi} \mathtt{P}$.

Theorem A.12 (Normalization for Natural Deduction)

If $\Gamma \vdash_{\Psi,\Phi} \mathtt{P}$ then $\Gamma \vdash_{\Psi,\Phi} \mathtt{P} \Uparrow$.

A special form of the above theorem is "if $[\![\mathtt{P}]\!]_{\Psi,\Phi}$ then $\cdot \vdash_{\Psi,\Phi} \mathtt{P} \Uparrow$". The soundness of *PropX* interpretation (Theorem A.1) can be proved using this theorem.

Appendix B

XCAP Soundness Proof

Due to the usage of simple semantic subsumptions ($\Rightarrow$ and $\Rightarrow_{\mathtt{c}}$, defined using logical implication $\supset$, as in Figure 3.3 and Figure 6.4), it is straight-forward to prove the soundness of XCAP inference rules using the syntactic soundness approach proposed by Wright and Felleisen [59]. In this section, we present the proof sketch for the soundness of XCAP, as discussed in Chapter 3, Chapter 4, and Chapter 5.

Lemma B.1 (XCAP Instruction Sequence Weakening)

If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$, $\Psi \subseteq \Psi'$, and $\mathtt{a}' \Rightarrow \mathtt{a}$ then $\Psi' \vdash \{\mathtt{a}'\}\,\mathbb{I}$.

Proof Sketch. The proof is by induction over the derivation $\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$, named as $\mathcal{D}$.

Case SEQ. The derivation $\mathcal{D}$ has the form

$$\frac{\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}'' \qquad \Psi \vdash \{\mathtt{a}''\}\,\mathbb{I}'}{\Psi \vdash \{\mathtt{a}\}\,\mathtt{c};\mathbb{I}'}$$

We have

1. from $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}''$ it follows that $\mathtt{a}' \Rightarrow_{\mathtt{c}} \mathtt{a}''$;

2. from $\Psi \vdash \{\mathtt{a}''\}\,\mathbb{I}'$ by the induction hypodissertation it follows that $\Psi' \vdash \{\mathtt{a}''\}\,\mathbb{I}'$.

Case JD. The derivation $\mathcal{D}$ has the form

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jd}\;\mathtt{f}}$$

From $\mathsf{a}\Rightarrow\mathsf{a}'$ and $\mathsf{a}\Rightarrow\Psi(\mathtt{f})$ it follows that $\mathsf{a}'\Rightarrow\Psi(\mathtt{f})$.

Case BGTI. The derivation $\mathcal{D}$ has the form

$$\frac{\begin{array}{c}(\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)\leq i\rangle \Wedge \mathsf{a}\ (\mathbb{H},\mathbb{R}))\Rightarrow\mathsf{a}'' \qquad \Psi\vdash\{\mathsf{a}''\}\,\mathbb{I}' \\ (\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)>i\rangle \Wedge \mathsf{a}\ (\mathbb{H},\mathbb{R}))\Rightarrow\Psi(\mathtt{f}) \quad \mathtt{f}\in\mathsf{dom}(\Psi)\end{array}}{\Psi\vdash\{\mathsf{a}\}\,\mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f};\mathbb{I}'}$$

We have

1. from $\mathsf{a}\Rightarrow\mathsf{a}'$ and $(\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)\leq i\rangle \Wedge \mathsf{a}\ (\mathbb{H},\mathbb{R}))\Rightarrow\mathsf{a}''$ it follows that $(\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)\leq i\rangle \Wedge \mathsf{a}'\ (\mathbb{H},\mathbb{R}))\Rightarrow\mathsf{a}''$;

2. from $\Psi\vdash\{\mathsf{a}''\}\,\mathbb{I}'$ and the induction hypodissertation it follows that $\Psi'\vdash\{\mathsf{a}''\}\,\mathbb{I}'$;

3. from $\mathsf{a}\Rightarrow\mathsf{a}'$ and $(\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)>i\rangle \Wedge \mathsf{a}\ (\mathbb{H},\mathbb{R}))\Rightarrow\Psi(\mathtt{f})$ it follows that $(\lambda(\mathbb{H},\mathbb{R}).\,\langle\mathbb{R}(\mathtt{r}_s)>i\rangle \Wedge \mathsf{a}'\ (\mathbb{H},\mathbb{R}))\Rightarrow\Psi(\mathtt{f})$.

Case JMP. The derivation $\mathcal{D}$ has the form

$$\frac{\mathsf{a}\Rightarrow(\lambda(\mathbb{H},\mathbb{R}).\,\mathsf{a}''\ (\mathbb{H},\mathbb{R})\ \Wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}''))}{\Psi\vdash\{\mathsf{a}\}\,\mathsf{jmp}\ \mathtt{r}}$$

From $\mathsf{a}\Rightarrow\mathsf{a}'$ and $\mathsf{a}\Rightarrow(\lambda(\mathbb{H},\mathbb{R}).\,\mathsf{a}''\ (\mathbb{H},\mathbb{R})\ \Wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}''))$ it follows that $\mathsf{a}'\Rightarrow(\lambda(\mathbb{H},\mathbb{R}).\,\mathsf{a}''\ (\mathbb{H},\mathbb{R})\ \Wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}''))$.

Case ECP. The derivation $\mathcal{D}$ has the form

$$\frac{(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f}))\Wedge\mathsf{a}\ \mathbb{S})\Rightarrow\mathsf{a}'' \qquad \mathtt{f}\in\mathsf{dom}(\Psi) \qquad \Psi\vdash\{\mathsf{a}''\}\,\mathbb{I}}{\Psi\vdash\{\mathsf{a}\}\,\mathbb{I}}$$

We have

1. from $\mathsf{a}\Rightarrow\mathsf{a}'$ and $(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f}))\Wedge\mathsf{a}\ \mathbb{S})\Rightarrow\mathsf{a}''$ it follows that $(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f}))\Wedge\mathsf{a}'\ \mathbb{S})\Rightarrow\mathsf{a}''$;

2. from $\Psi\vdash\{\mathsf{a}''\}\,\mathbb{I}$ by the induction hypodissertation it follows that $\Psi'\vdash\{\mathsf{a}''\}\,\mathbb{I}$. ■

Lemma B.2 (XCAP Code Heap Typing)

If $\Psi_{IN}\vdash\mathbb{C}:\Psi$ and $\mathtt{f}\in\mathsf{dom}(\Psi)$ then $\mathtt{f}\in\mathsf{dom}(\mathbb{C})$ and $\Psi_{IN}\vdash\{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

Proof Sketch. The proof is by induction over the derivation $\Psi_{IN}\vdash\mathbb{C}:\Psi$.

Case CDHP. The derivation has the form

$$\frac{\Psi_{IN} \vdash \{\mathtt{a}_i\}\,\mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \rightsquigarrow \mathbb{I}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathbb{I}_n\} : \{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\}}$$

From $\mathtt{f} \in \mathsf{dom}(\Psi)$ it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

Case LINK. The derivation has the form

$$\frac{\begin{array}{c}\Psi_{IN\,1} \vdash \mathbb{C}_1 : \Psi_1 \qquad \Psi_{IN\,2} \vdash \mathbb{C}_2 : \Psi_2 \qquad \Psi_{IN\,1}(\mathtt{f}) = \Psi_{IN\,2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN\,1}) \cap \mathsf{dom}(\Psi_{IN\,2})\end{array}}{\Psi_{IN\,1} \cup \Psi_{IN\,2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2}$$

From $\mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset$ it follows that $\mathsf{dom}(\Psi_1) \cap \mathsf{dom}(\Psi_2) = \emptyset$. Then from $\mathtt{f} \in \mathsf{dom}(\Psi)$ it follows that $\mathtt{f} \in \mathsf{dom}(\Psi_i)$, where i could be either 1 or 2. From $\Psi_{IN\,i} \vdash \mathbb{C}_i : \Psi_i$ by the induction hypodissertation it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C}_i)$ and $\Psi_{IN\,i} \vdash \{\Psi_i(\mathtt{f})\}\,\mathbb{C}_i(\mathtt{f})$.

From $\mathtt{f} \in \mathsf{dom}(\mathbb{C}_i)$ it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C}_1 \cup \mathbb{C}_2)$.

From $\Psi_{IN\,i} \vdash \{\Psi_i(\mathtt{f})\}\,\mathbb{C}_i(\mathtt{f})$ and Instruction Sequence Weakening (Lemma B.1) it follows that $\Psi_{IN\,1} \cup \Psi_{IN\,2} \vdash \{\Psi_1 \cup \Psi_2(\mathtt{f})\}\,\mathbb{C}_1 \cup \mathbb{C}_2(\mathtt{f})$. ■

Lemma B.3 (XCAP State Typing)

If $\Psi_{IN} \vdash \mathbb{C} : \Psi$ and $[\![\mathsf{cptr}(\mathtt{f}, \mathtt{a})]\!]_{\Psi}$ then $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\mathtt{a}\}\,\mathbb{C}(\mathtt{f})$.

Proof Sketch. From $[\![\mathsf{cptr}(\mathtt{f}, \mathtt{a})]\!]_{\Psi}$ by *PropX* Validity Soundness (Lemma A.1) it follows that $\mathtt{f} \in \mathsf{dom}(\Psi)$ and $\Psi(\mathtt{f}) = \mathtt{a}$. By Code Heap Typing (Lemma B.2) it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\mathtt{a}\}\,\mathbb{C}(\mathtt{f})$. ■

Lemma B.4 (XCAP Progress)

If $\Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Proof Sketch. Derivation $\Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{P}$ has the following form.

$$\frac{\Psi_{G} \vdash \mathbb{C} : \Psi_{G} \qquad ([\![\mathtt{a}]\!]_{\Psi_{G}}\ \mathbb{S}) \qquad \Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi_{G} \vdash \{\mathtt{a}\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})}$$

The proof is by induction over derivation $\Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{I}$.

Case SEQ, JD, BGTI and JMP. the proof is by simple inspections.

Case ECP. The proof is by the induction hypodissertation. ■

Lemma B.5 (XCAP Preservation)

If $\Psi_{G} \vdash \{\mathtt{a}\}\,\mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$ then there exists an assertion $\mathtt{a}'$ such that $\Psi_{G} \vdash \{\mathtt{a}'\}\,\mathbb{P}'$.

Proof Sketch. Suppose $\mathbb{P} = (\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I})$. We name derivation $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$ as $\mathcal{D}$, which has the following form.

$$\frac{\Psi_G \vdash \mathbb{C} : \Psi_G \qquad ([\![\,\mathtt{a}\,]\!]_{\Psi_G}\ (\mathbb{H}, \mathbb{R})) \qquad \Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi_G \vdash \{\mathtt{a}\}\,(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I})}$$

The proof is by induction over the derivation $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}$, named as $\mathcal{E}$.

Case SEQ. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' \qquad \Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}'}{\Psi_G \vdash \{\mathtt{a}\}\,\mathtt{c}; \mathbb{I}'}$$

By the operational semantics, $\mathbb{P}' = (\mathbb{C}, \mathtt{Next}_{\mathtt{c}}(\mathbb{H}, \mathbb{R}), \mathbb{I}')$. Then

1. $\Psi_G \vdash \mathbb{C} : \Psi_G$ is in $\mathcal{D}$;
2. from $([\![\,\mathtt{a}\,]\!]_{\Psi_G}\ (\mathbb{H}, \mathbb{R}))$ and $\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}'$ it follows that $([\![\,\mathtt{a}'\,]\!]_{\Psi_G}\ \mathtt{Next}_{\mathtt{c}}(\mathbb{H}, \mathbb{R}))$;
3. $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}'$ is in $\mathcal{E}$.

Case JD. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow \Psi_G(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi_G)}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{jd}\ \mathtt{f}}$$

By the operational semantics, $\mathbb{P}' = (\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$. Then

1. $\Psi_G \vdash \mathbb{C} : \Psi_G$ is in $\mathcal{D}$;
2. from $([\![\,\mathtt{a}\,]\!]_{\Psi_G}\ (\mathbb{H}, \mathbb{R}))$ and $\mathtt{a} \Rightarrow \Psi_G(\mathtt{f})$ it follows that $([\![\,\Psi_G(\mathtt{f})\,]\!]_{\Psi_G}\ (\mathbb{H}, \mathbb{R}))$;
3. by $\Psi_G \vdash \mathbb{C} : \Psi_G$, $\mathtt{f} \in \mathsf{dom}(\Psi_G)$, and Code Heap Typing (Lemma B.2) it follows that $\Psi_G \vdash \{\Psi_G(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

Case BGTI. The derivation $\mathcal{E}$ has the form

$$\frac{\begin{array}{ll}(\lambda(\mathbb{H}, \mathbb{R}).\, \langle \mathbb{R}(\mathtt{r}_s) \le i \rangle \ \wedge\!\!\!\wedge\ \mathtt{a}\ (\mathbb{H}, \mathbb{R})) \Rightarrow \mathtt{a}' & \Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}' \\ (\lambda(\mathbb{H}, \mathbb{R}).\, \langle \mathbb{R}(\mathtt{r}_s) > i \rangle \ \wedge\!\!\!\wedge\ \mathtt{a}\ (\mathbb{H}, \mathbb{R})) \Rightarrow \Psi_G(\mathtt{f}) & \mathtt{f} \in \mathsf{dom}(\Psi_G)\end{array}}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}'}$$

By the operational semantics, when $\mathbb{R}(\mathtt{r}_s) > i$, $\mathbb{P}' = (\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$. Then

1. $\Psi_G \vdash \mathbb{C} : \Psi_G$ is in $\mathcal{D}$;

2. from $([\![\mathtt{a}]\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$ and $(\lambda(\mathbb{H},\mathbb{R}).\,\langle \mathbb{R}(\mathtt{r}_s) \le i \rangle \ \wedge\!\!\!\wedge\ \mathtt{a}\ (\mathbb{H},\mathbb{R})) \Rightarrow \mathtt{a}'$ it follows that $([\![\Psi_G(\mathtt{f})]\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$;

3. by $\Psi_G \vdash \mathbb{C} : \Psi_G$, $\mathtt{f} \in \mathsf{dom}(\Psi_G)$, and Code Heap Typing (Lemma B.2) it follows that $\Psi_G \vdash \{\Psi_G(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

By the operational semantics, when $\mathbb{R}(\mathtt{r}_s) \le i$, $\mathbb{P}' = (\mathbb{C}, (\mathbb{H},\mathbb{R}), \mathbb{I}')$. Then

1. $\Psi_G \vdash \mathbb{C} : \Psi_G$ is in $\mathcal{D}$;

2. from $([\![\mathtt{a}]\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$ and $(\lambda(\mathbb{H},\mathbb{R}).\,\langle \mathbb{R}(\mathtt{r}_s) \le i \rangle \ \wedge\!\!\!\wedge\ \mathtt{a}\ (\mathbb{H},\mathbb{R})) \Rightarrow \mathtt{a}'$ it follows that $([\![\mathtt{a}']\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$;

3. $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}'$ is in $\mathcal{E}$.

Case JMP. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\,\mathtt{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\!\!\!\wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}'))}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}$$

From $([\![\mathtt{a}]\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$ and $\mathtt{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\,\mathtt{a}'\ (\mathbb{H},\mathbb{R})\ \wedge\!\!\!\wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}'))$ it follows that $[\![\mathtt{a}']\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R})$ and $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}')]\!]_{\Psi_G}$.
By the operational semantics, $\mathbb{P}' = (\mathbb{C}, (\mathbb{H},\mathbb{R}), \mathbb{C}(\mathbb{R}(\mathtt{r})))$. Then

1. $\Psi_G \vdash \mathbb{C} : \Psi_G$ is in $\mathcal{D}$;

2. $([\![\mathtt{a}']\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$ is from above;

3. by $\Psi_G \vdash \mathbb{C} : \Psi_G$, $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}), \mathtt{a}')]\!]_{\Psi_G}$, and State Typing (Lemma B.3) it follows that $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{C}(\mathbb{R}(\mathtt{r}))$.

Case ECP. The derivation $\mathcal{E}$ has the form

$$\frac{(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f}, APP\Psi_G\mathtt{f}) \wedge\!\!\!\wedge \mathtt{a}\ \mathbb{S}) \Rightarrow \mathtt{a}' \qquad \mathtt{f} \in \mathsf{dom}(\Psi_G) \qquad \Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}}{\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}}$$

From $([\![\texttt{a}]\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$, $(\lambda\mathbb{S}.\,\mathsf{cptr}(\texttt{f},\Psi_G(\texttt{f}))\mathrel{\wedge\!\!\!\wedge}\texttt{a}\ \mathbb{S})\Rightarrow\texttt{a}'$, and $\texttt{f}\in\mathsf{dom}(\Psi_G)$ it follows that $([\![\texttt{a}']\!]_{\Psi_G}\ (\mathbb{H},\mathbb{R}))$. Together with $\Psi_G\vdash\{\texttt{a}'\}\,\mathbb{I}$, the prove is completed by using the induction hypodissertation. ■

Theorem B.6 (XCAP Soundness)

If $\Psi_G\vdash\{\texttt{a}\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P}\longmapsto^{n}\mathbb{P}'$.

Proof Sketch. By simple induction on n and using Progress (Lemma B.4) and Perservation (Lemma B.5). ■

Appendix C

TAL to XCAP Translation Typing Preservation Proof

In this section, we present the proof for the typing preservation theorem of the translation from TAL to XCAP in Chapter 5. Here TAL is as defined in Section 5.2, not the "semantic" one in Section 5.3.

Lemma C.1 (Value Typing Preservation)

If $\Psi;\Phi \vdash_{\text{TAL}} \mathtt{w} : \tau$ then $[\![\ulcorner\tau\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$.

Proof Sketch. By induction on derivation $\Psi;\Phi \vdash_{\text{TAL}} \mathtt{w} : \tau$.

Case INT. The derivation has the form

$$\frac{}{\Psi;\Phi \vdash_{\text{TAL}} \mathtt{w} : \mathsf{int}}$$

We have $\ulcorner\mathsf{int}\urcorner = \lambda\mathtt{w}.\,\mathsf{True}$, Trivial.

Case CODE. The derivation has the form

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi;\Phi \vdash_{\text{TAL}} \mathtt{f} : \mathsf{code}\ \ \Psi(\mathtt{f})}$$

We have $\ulcorner\mathsf{code}\ \ \Psi(\mathtt{f})\urcorner = \lambda\mathtt{w}.\,\mathsf{codeptr}(\mathtt{w},\ulcorner\Psi(\mathtt{f})\urcorner) = \lambda\mathtt{w}.\,\mathsf{codeptr}(\mathtt{w},\ulcorner\Psi\urcorner(\mathtt{f}))$, Trivial.

Case POLY. The derivation has the form

$$\frac{\cdot \vdash \tau' \qquad \Psi;\Phi \vdash_{\text{TAL}} \mathtt{f} : \mathsf{code}\ [\alpha,\Delta].\,\Gamma}{\Psi;\Phi \vdash_{\text{TAL}} \mathtt{f} : \mathsf{code}\ [\Delta].\,\Gamma[\tau'/\alpha]}$$

We have $\ulcorner\mathsf{code}\ [\Delta].\Gamma[\tau'/\alpha]\urcorner = \lambda\mathtt{w}.\,\mathsf{codeptr}(\mathtt{w},\ulcorner[\Delta].\Gamma[\tau'/\alpha]\urcorner) = \lambda\mathtt{w}.\,\mathsf{codeptr}(\mathtt{w},\exists\Delta.\ulcorner\Gamma[\tau'/\alpha]\urcorner)$.

By the induction hypodissertation it follows that $[\![\ulcorner\mathsf{code}\ [\alpha,\Delta].\Gamma\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$, and then

$[\![\mathsf{codeptr}(\mathtt{w},\exists\alpha,\Delta.\ulcorner\Gamma\urcorner)]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$.

It is easy to show $[\![\forall\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma[\tau'/\alpha]\urcorner\ \mathbb{S} \twoheadrightarrow \exists\alpha,\Delta.\ulcorner\Gamma\urcorner\ \mathbb{S}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. IT then follows that

$[\![\mathsf{codeptr}(\mathtt{w},\exists\Delta.\ulcorner\Gamma[\tau'/\alpha]\urcorner)]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$.

Case TUPLE. The derivation has the form

$$\frac{\Phi(\mathtt{l}+i-1)=\tau_i \qquad \forall\, i}{\Psi;\Phi\vdash_{\mathrm{TAL}}\mathtt{l}:\langle\tau_1,\ldots,\tau_n\rangle}$$

$$\begin{aligned}\text{We have}\quad \ulcorner\langle\tau_1,\ldots,\tau_n\rangle\urcorner &= \lambda\mathtt{w}.\,\mathsf{record}(\mathtt{w},\ulcorner\tau_1\urcorner,\ldots,\ulcorner\tau_n\urcorner)\\ &= \lambda\mathtt{w}.\,\mathsf{ref}(\mathtt{w},\ulcorner\tau_1\urcorner) \mathbin{\&\!\!\&} \ldots \mathbin{\&\!\!\&} \mathsf{ref}(\mathtt{w}+n-1,\ulcorner\tau_n\urcorner).\\ &= \lambda\mathtt{w}.\,\mathsf{ref}(\mathtt{w},\ulcorner\Phi(\mathtt{w})\urcorner) \mathbin{\&\!\!\&} \ldots \mathbin{\&\!\!\&} \mathsf{ref}(\mathtt{w}+n-1,\ulcorner\Phi(\mathtt{w}+n-1)\urcorner).\\ &= \lambda\mathtt{w}.\,\mathsf{ref}(\mathtt{w},\ulcorner\Phi\urcorner(\mathtt{w})) \mathbin{\&\!\!\&} \ldots \mathbin{\&\!\!\&} \mathsf{ref}(\mathtt{w}+n-1,\ulcorner\Phi\urcorner(\mathtt{w}+n-1)).\end{aligned}$$

Trivial.

Case EXT. The derivation has the form

$$\frac{\cdot\vdash\tau' \qquad \Psi;\Phi\vdash_{\mathrm{TAL}}\mathtt{w}:\tau[\tau'/\alpha]}{\Psi;\Phi\vdash_{\mathrm{TAL}}\mathtt{w}:\exists\alpha.\,\tau}$$

We have $\ulcorner\exists\alpha.\,\tau\urcorner = \lambda\mathtt{w}.\,\exists\,\alpha.\ulcorner\tau\urcorner\ \mathtt{w}$. By the induction hypodissertation it follows that

$[\![\ulcorner\tau[\tau'/\alpha]\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. It then follows that $[\![\exists\,\alpha.\ulcorner\tau\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$.

Case ALL. The derivation has the form

$$\frac{\Psi;\Phi\vdash_{\mathrm{TAL}}\mathtt{w}:\tau[\mu\alpha.\,\tau/\alpha]}{\Psi;\Phi\vdash_{\mathrm{TAL}}\mathtt{w}:\mu\alpha.\,\tau}$$

We have $\ulcorner\mu\alpha.\,\tau\urcorner = \lambda\mathtt{w}.\,(\boldsymbol{\mu}\,\alpha.\,\lambda x.\,(\ulcorner\tau\urcorner\ x)\quad \mathtt{w})$. By the induction hypodissertation it follows that $[\![\ulcorner\tau[\mu\alpha.\,\tau/\alpha]\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. It then follows that $[\![(\boldsymbol{\mu}\,\alpha.\,\lambda x.\,(\ulcorner\tau\urcorner\ x)\quad \mathtt{w})]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. ■

Lemma C.2 (Register File Typing Preservation)

If $\Psi;\Phi\vdash_{\mathrm{TAL}}\mathbb{R}:\Gamma$ then $[\![\ulcorner[\,].\Gamma\urcorner\ (\mathbb{H},\mathbb{R})]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$.

Proof Sketch. Derivation $\Psi;\Phi\vdash_{\mathrm{TAL}}\mathbb{R}:\Gamma$ has the form

$$\frac{\Psi;\Phi\vdash_{\mathrm{TAL}}\mathbb{R}(\mathtt{r}_i):\tau_i \qquad \forall\, i}{\Psi;\Phi\vdash_{\mathrm{TAL}}\mathbb{R}:\{\mathtt{r}_1\rightsquigarrow\tau_1,\ldots,\mathtt{r}_n\rightsquigarrow\tau_n\}}$$

We have $\ulcorner [].\Gamma \urcorner \;=\; \lambda(\mathbb{H},\mathbb{R}).\,(\ulcorner \tau_1 \urcorner\, \mathbb{R}(\mathtt{r}_1)) \mathbin{\wedge\!\!\!\wedge} \ldots \mathbin{\wedge\!\!\!\wedge} (\ulcorner \tau_n \urcorner\, \mathbb{R}(\mathtt{r}_n))$.

From $\Psi;\Phi \vdash_{\text{TAL}} \mathbb{R}(\mathtt{r}_i) : \tau_i$ by Value Typing Preservation (Lemma C.1) it follows that $\llbracket \ulcorner \tau_i \urcorner\, \mathbb{R}(\mathtt{r}_i) \rrbracket_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. ■

Lemma C.3 (Data Heap Typing Preservation)

If $\Psi \vdash_{\text{TAL}} \mathbb{H} : \Phi$ then $\mathcal{DH}\; \ulcorner\Psi\urcorner\; \ulcorner\Phi\urcorner\; \mathbb{H}$.

Proof Sketch. Derivation $\Psi \vdash_{\text{TAL}} \mathbb{H} : \Phi$ has the form

$$\frac{\Psi;\Phi \vdash_{\text{TAL}} \mathbb{H}(\mathtt{l}) : \Phi(\mathtt{l}) \qquad \forall\, \mathtt{l} \in \mathsf{dom}(\Phi) = \mathsf{dom}(\mathbb{H})}{\Psi \vdash_{\text{TAL}} \mathbb{H} : \Phi}$$

We have $\mathcal{DH}\; \ulcorner\Psi\urcorner\; \ulcorner\Phi\urcorner\; \mathbb{H} \;=\; \forall \mathtt{l} \in \mathsf{dom}(\Phi) = \mathsf{dom}(\mathbb{H}).\; \llbracket \Phi(\mathtt{l})\; \mathbb{H}(\mathtt{l}) \rrbracket_{\Psi,\Phi}$.

From $\Psi;\Phi \vdash_{\text{TAL}} \mathbb{H}(\mathtt{l}) : \Phi(\mathtt{l})$ by Value Typing Preservation (Lemma C.1) it follows that $\llbracket \ulcorner \Phi(\mathtt{l}) \urcorner\; \mathbb{H}(\mathtt{l}) \rrbracket_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. ■

Lemma C.4 (State Typing Preservation)

If $\Psi \vdash_{\text{TAL}} \mathbb{S} : [\Delta].\,\Gamma$ then $\llbracket \ulcorner [\Delta].\,\Gamma \urcorner \rrbracket_{\ulcorner\Psi\urcorner}\; \mathbb{S}$.

Proof Sketch. Derivation $\Psi \vdash_{\text{TAL}} \mathbb{S} : [\Delta].\,\Gamma$ has the form

$$\frac{\cdot \vdash \tau_i \qquad \forall\, i \qquad \Psi \vdash_{\text{TAL}} \mathbb{H} : \Phi \qquad \Psi;\Phi \vdash_{\text{TAL}} \mathbb{R} : \Gamma[\tau_1,\ldots,\tau_n/\alpha_1,\ldots,\alpha_n]}{\Psi \vdash_{\text{TAL}} (\mathbb{H},\mathbb{R}) : [\alpha_1,\ldots,\alpha_n].\Gamma}$$

From $\Psi \vdash_{\text{TAL}} \mathbb{H} : \Phi$ by Data Heap Typing Preservation (Lemma C.3) it follows that $\mathcal{DH}\; \ulcorner\Psi\urcorner\; \ulcorner\Phi\urcorner\; \mathbb{H}$.

From $\Psi;\Phi \vdash_{\text{TAL}} \mathbb{R} : \Gamma[\tau_1,\ldots,\tau_n/\alpha_1,\ldots,\alpha_n]$ by Register File Typing Preservation (Lemma C.2) it follows that $\llbracket \ulcorner \Gamma[\tau_1,\ldots,\tau_n/\alpha_1,\ldots,\alpha_n] \urcorner\; (\{\},\mathbb{R}) \rrbracket_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. It then follows that $\llbracket \ulcorner [\alpha_1,\ldots,\alpha_n].\Gamma \urcorner\; (\{\},\mathbb{R}) \rrbracket_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$.

Finally, it follows that $\llbracket \ulcorner [\alpha_1,\ldots,\alpha_n].\Gamma \urcorner \rrbracket_{\ulcorner\Psi\urcorner}\; (\mathbb{H},\mathbb{R})$. ■

Lemma C.5 (Subtyping Preservation)

If $\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'$ then $\ulcorner [\Delta].\,\Gamma \urcorner \Rightarrow \ulcorner [\Delta'].\,\Gamma' \urcorner$.

Proof Sketch. By casing derivation $\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'$.

Case SUBT. The derivation has the form

$$\frac{\Delta \supseteq \Delta' \qquad \forall\, \mathtt{r} \in \mathsf{dom}(\Gamma') \qquad \Gamma(\mathtt{r}) = \Gamma'(\mathtt{r}) \qquad \Delta' \vdash \Gamma'(\mathtt{r})}{\vdash_{\text{TAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'}$$

We have $\ulcorner[\Delta].\Gamma\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma\urcorner\,\mathbb{S}$ and $\ulcorner[\Delta'].\Gamma'\urcorner = \lambda\mathbb{S}.\,\exists\Delta'.\ulcorner\Gamma'\urcorner\,\mathbb{S}$. Trivial.

Case TAPP. The derivation has the form

$$\frac{\Gamma(\mathtt{r}) = \mathsf{code}\ [\alpha,\Delta'].\,\Gamma' \qquad \Delta \vdash \tau'}{\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \mathsf{code}\ [\Delta'].\,\Gamma'[\tau'/\alpha]\}}$$

Suppose $\Gamma = \Gamma'' \cup \{\mathtt{r} \leadsto \mathsf{code}\ [\alpha,\Delta'].\,\Gamma'\}$.

We have $\ulcorner[\Delta].\Gamma\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma\urcorner\,\mathbb{S} = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}), \exists\,\alpha,\Delta'.\,\Gamma')$ and

$\ulcorner\mathsf{code}\ [\Delta'].\,\Gamma'[\tau'/\alpha]\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma\urcorner\,\mathbb{S} = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}), \exists\Delta'.\,\Gamma'[\tau'/\alpha])$.

It is easy to show $\llbracket\forall\mathbb{S}.\,\exists\Delta'.\ulcorner\Gamma'[\tau'/\alpha]\urcorner\,\mathbb{S} \twoheadrightarrow \exists\,\alpha,\Delta'.\ulcorner\Gamma'\urcorner\,\mathbb{S}\rrbracket_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$. Trivial.

Case PACK. The derivation has the form

$$\frac{\Gamma(\mathtt{r}) = \tau[\tau'/\alpha] \qquad \Delta \vdash \tau'}{\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \exists\alpha.\,\tau\}}$$

Suppose $\Gamma = \Gamma'' \cup \{\mathtt{r} \leadsto \tau[\tau'/\alpha]\}$.

We have $\ulcorner[\Delta].\Gamma\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \ulcorner\tau[\tau'/\alpha]\urcorner\,\mathbb{S}.\mathbb{R}(\mathtt{r})$ and

$\ulcorner[\Delta].\Gamma\{\mathtt{r} : \exists\alpha.\,\tau\}\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \exists\,\alpha.\ulcorner\tau\urcorner\,\mathbb{S}.\mathbb{R}(\mathtt{r})$. Trivial.

Case UNPACK. The derivation has the form

$$\frac{\Gamma(\mathtt{r}) = \exists\alpha.\,\tau}{\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\alpha,\Delta].\Gamma\{\mathtt{r} : \tau\}}$$

Suppose $\Gamma = \Gamma'' \cup \{\mathtt{r} \leadsto \exists\alpha.\,\tau\}$.

We have $\ulcorner[\Delta].\Gamma\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \exists\,\alpha.\ulcorner\tau\urcorner\,\mathbb{S}.\mathbb{R}(\mathtt{r})$ and

$\ulcorner[\alpha,\Delta].\Gamma\{\mathtt{r} : \tau\}\urcorner = \lambda\mathbb{S}.\,\exists\,\alpha,\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \ulcorner\tau[\tau'/\alpha]\urcorner\,\mathbb{S}.\mathbb{R}(\mathtt{r})$. Trivial.

Case FOLD. The derivation has the form

$$\frac{\Gamma(\mathtt{r}) = \tau[\mu\alpha.\,\tau/\alpha]}{\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \mu\alpha.\,\tau\}}$$

Suppose $\Gamma = \Gamma'' \cup \{\mathtt{r} \leadsto \tau[\mu\alpha.\,\tau/\alpha]\}$.

We have $\ulcorner[\Delta].\Gamma\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \ulcorner\tau[\mu\alpha.\,\tau/\alpha]\urcorner\,\mathbb{S}.\mathbb{R}(\mathtt{r})$ and

$\ulcorner[\Delta].\Gamma\{\mathtt{r} : \mu\alpha.\,\tau\}\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge (\boldsymbol{\mu}\,\alpha.\,\lambda x.\ulcorner\tau\urcorner\,x\ \ \mathbb{S}.\mathbb{R}(\mathtt{r}))$. Trivial.

Case UNFOLD. The derivation has the form

$$\frac{\Gamma(\mathtt{r}) = \mu\alpha.\,\tau}{\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r}:\tau[\mu\alpha.\,\tau/\alpha]\}}$$

Suppose $\Gamma = \Gamma'' \cup \{\mathtt{r} \leadsto \mu\alpha.\,\tau\}$.

We have $\ulcorner[\Delta].\Gamma\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge (\boldsymbol{\mu}\,\alpha.\,\lambda x.\ulcorner\tau\urcorner\,x\ \ \mathbb{S}.\mathbb{R}(\mathtt{r}))$ and

$\ulcorner\Gamma\{\mathtt{r}:\tau[\mu\alpha.\,\tau/\alpha]\}\urcorner = \lambda\mathbb{S}.\,\exists\Delta.\ulcorner\Gamma''\urcorner\,\mathbb{S} \wedge \ulcorner\tau[\mu\alpha.\,\tau/\alpha]\urcorner\,\mathbb{S}.\mathbb{R}(\mathtt{r})$. Trivial. ■

Lemma C.6 (Instruction Typing Preservation)

If $\Psi \vdash_{\text{TAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\ulcorner[\Delta].\,\Gamma\urcorner_\Psi \Rightarrow_{\mathsf{c}} \ulcorner[\Delta].\,\Gamma'\urcorner$.

Proof Sketch. By casing on derivation $\Psi \vdash_{\text{TAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$. Trivial. ■

Lemma C.7 (Instruction Sequence Typing Preservation)

If $\Psi \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathbb{I}$ then $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathbb{I}$.

Proof Sketch. By induction over derivation $\Psi \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathbb{I}$.

case WEAKEN. The derivation has the form

$$\frac{\vdash_{\text{TAL}} [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma' \qquad \Psi \vdash_{\text{TAL}} \{[\Delta'].\,\Gamma'\}\mathbb{I}}{\Psi \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathbb{I}}$$

By induction hypodissertation it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta'].\,\Gamma'\urcorner\}\mathbb{I}$. By Subtyping Preservation (Lemma C.5) it follows that $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\,\Gamma'\urcorner$. The by XCAP Instruction Sequence Weakening (Lemma B.1) it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathbb{I}$.

case SEQ. The derivation has the form

$$\frac{\Psi \vdash_{\text{TAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\} \qquad \Psi \vdash_{\text{TAL}} \{[\Delta].\,\Gamma'\}\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{mov}, \mathsf{movi}, \mathsf{ld}, \mathsf{st}\}}{\Psi \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathtt{c};\mathbb{I}}$$

By induction hypodissertation it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma'\urcorner\}\mathbb{I}$. By Instruction Typing Preservation (Lemma C.6) it follows that $\ulcorner[\Delta].\,\Gamma\urcorner_\Psi \Rightarrow_{\mathsf{c}} \ulcorner[\Delta].\,\Gamma'\urcorner$. By XCAP rule SEQ it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathtt{c};\mathbb{I}$.

case JD. The derivation has the form

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \vdash_{\text{TAL}} [\Delta].\,\Gamma \leq \Psi(\mathtt{f})}{\Psi \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathsf{jd}\ \mathtt{f}}$$

By Subtyping Preservation (Lemma C.5) it follows that $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \ulcorner\Psi(\mathtt{f})\urcorner$, and then $\ulcorner[\Delta].\,\Gamma\urcorner \Rightarrow \ulcorner\Psi\urcorner(\mathtt{f})$. By XCAP rule JD it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\,\Gamma\urcorner\}\mathsf{jd}\ \mathtt{f}$.

case JMP. The derivation has the form

$$\frac{\Gamma(\mathtt{r}) = \mathsf{code}\ [\Delta'].\Gamma' \qquad \vdash_{\text{TAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'}{\Psi \vdash_{\text{TAL}} \{[\Delta].\Gamma\}\mathsf{jmp}\ \mathtt{r}}$$

By translation it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \lambda(\mathbb{H},\mathbb{R}).\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\ulcorner[\Delta'].\Gamma'\urcorner)$. By Subtyping Preservation (Lemma C.5) it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\Gamma'\urcorner$. By XCAP rule JMP it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathsf{jmp}\ \mathtt{r}$.

case MOVF. The derivation has the form

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash_{\text{TAL}} \{[\Delta].\Gamma\{\mathtt{r}_d \leadsto \mathsf{code}\ \ \Psi(\mathtt{f})\}\}\mathbb{I}}{\Psi \vdash_{\text{TAL}} \{[\Delta].\Gamma\}\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}}$$

By induction hypodissertation it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\Gamma\{\mathtt{r}_d \leadsto \mathsf{code}\ \ \Psi(\mathtt{f})\}\urcorner\}\mathbb{I}$. By XCAP rule ECP and SEQ it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}$. ■

Lemma C.8 (Code Heap Typing Preservation)

If $\Psi_{IN} \vdash_{\text{TAL}} \mathbb{C}:\Psi$ then $\ulcorner\Psi_{IN}\urcorner \vdash_{\text{XCAP}} \mathbb{C}:\ulcorner\Psi\urcorner$.

Proof Sketch. By induction on derivation $\Psi_{IN} \vdash_{\text{TAL}} \mathbb{C}:\Psi$.

case CDHP. The derivation has the form

$$\frac{\Psi_{IN} \vdash_{\text{TAL}} \{\Psi(\mathtt{f})\}\mathbb{C}(\mathtt{f}) \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi_{IN} \vdash_{\text{TAL}} \mathbb{C}:\Psi}$$

From $\Psi_{IN} \vdash_{\text{TAL}} \{\Psi(\mathtt{f})\}\mathbb{C}(\mathtt{f})$ by Instruction Sequence Typing Preservation (Lemma C.7) it follows that $\ulcorner\Psi_{IN}\urcorner \vdash_{\text{XCAP}} \{\ulcorner\Psi\urcorner(\mathtt{f})\}\mathbb{C}(\mathtt{f})$.

case LINK. The derivation has the form

$$\frac{\begin{array}{ccc}\Psi_{IN1} \vdash_{\text{TAL}} \mathbb{C}_1:\Psi_1 & \Psi_{IN2} \vdash_{\text{TAL}} \mathbb{C}_2:\Psi_2 & \Psi_{IN1}(\mathtt{f}) = \Psi_{IN2}(\mathtt{f}) \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset & & \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN1}) \cap \mathsf{dom}(\Psi_{IN2})\end{array}}{\Psi_{IN1} \cup \Psi_{IN2} \vdash_{\text{TAL}} \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2}$$

From $\Psi_{INi} \vdash_{\text{TAL}} \mathbb{C}_i:\Psi_i$ by Instruction Sequence Typing Preservation (Lemma C.7) it follows that $\ulcorner\Psi_{INi}\urcorner \vdash_{\text{XCAP}} \mathbb{C}_i:\ulcorner\Psi_i\urcorner$. By XCAP rule LINK it follows that $\ulcorner\Psi_{IN1} \cup \Psi_{IN2}\urcorner \vdash_{\text{XCAP}} \mathbb{C}_1 \cup \mathbb{C}_2 : \ulcorner\Psi_1 \cup \Psi_2\urcorner$. ■

Theorem C.9 (Program Typing Preservation)

If $\Psi \vdash_{\text{TAL}} \{[\Delta].\Gamma\}\,\mathbb{P}$ then $\ulcorner\Psi\urcorner \vdash_{\text{XCAP}} \{\ulcorner[\Delta].\Gamma\urcorner\}\,\mathbb{P}$.

Proof Sketch. Derivation $\Psi \vdash_{\text{TAL}} \{[\Delta].\Gamma\}\,\mathbb{P}$ has the form

$$\frac{\Psi_G \vdash_{\text{TAL}} \mathbb{C} : \Psi_G \qquad \Psi_G \vdash_{\text{TAL}} \mathbb{S} : [\Delta].\,\Gamma \qquad \Psi_G \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathbb{I}}{\Psi_G \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})}$$

From $\Psi_G \vdash_{\text{TAL}} \mathbb{C} : \Psi_G$ by Code Heap Typing Preservation (Lemma C.8) it follows that $\ulcorner\Psi_G\urcorner \vdash_{\text{XCAP}} \mathbb{C} : \ulcorner\Psi_G\urcorner$.

From $\Psi_G \vdash_{\text{TAL}} \mathbb{S} : [\Delta].\,\Gamma$ by State Typing Preservation (Lemma C.4) it follows that $[\![\ulcorner[\Delta].\,\Gamma\urcorner]\!]_{\ulcorner\Psi_G\urcorner}\ \mathbb{S}$.

From $\Psi_G \vdash_{\text{TAL}} \{[\Delta].\,\Gamma\}\mathbb{I}$ by Instruction Sequence Typing Preservation (Lemma C.7) it follows that $\Psi_G \vdash_{\text{XCAP}} \{[\Delta].\,\Gamma\}\mathbb{I}$. ■

Appendix D

XCAP86 Soundness Proof

Following the soundness proof of the XCAP in Appendix B, we present the soundness proof of XCAP86 discussed in Chapter 6.

Lemma D.1 (XCAP86 Instruction Sequence Weakening)

If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$, $\Psi \subseteq \Psi'$, and $\mathtt{a}' \Rightarrow \mathtt{a}$ then $\Psi' \vdash \{\mathtt{a}'\}\,\mathbb{I}$.

Proof Sketch. The proof is by induction over the derivation $\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$, named as $\mathcal{D}$.

Case SEQ. The derivation $\mathcal{D}$ has the form

$$\frac{\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}'' \qquad \Psi \vdash \{\mathtt{a}''\}\,\mathbb{I}'}{\Psi \vdash \{\mathtt{a}\}\,\mathtt{c};\mathbb{I}'}$$

We have

1. from $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}''$ it follows that $\mathtt{a}' \Rightarrow_{\mathtt{c}} \mathtt{a}''$;

2. from $\Psi \vdash \{\mathtt{a}''\}\,\mathbb{I}'$ by the induction hypodissertation it follows that $\Psi' \vdash \{\mathtt{a}''\}\,\mathbb{I}'$.

Case JMPI. The derivation $\mathcal{D}$ has the form

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{f}}$$

From $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\mathtt{a} \Rightarrow \Psi(\mathtt{f})$ it follows that $\mathtt{a}' \Rightarrow \Psi(\mathtt{f})$.

Case JCC. The derivation $\mathcal{D}$ has the form

$$\frac{\begin{array}{cc}(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\neg\hat{\mathbb{F}}(cc)\rangle \curlywedge \mathsf{a}\ (\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \mathsf{a}'' & \Psi \vdash \{\mathsf{a}''\}\,\mathbb{I}' \\ (\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\hat{\mathbb{F}}(cc)\rangle \curlywedge \mathsf{a}\ (\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \Psi(\mathtt{f}) & \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{j}cc\ \mathtt{f};\mathbb{I}'}$$

We have

1. from $\mathsf{a} \Rightarrow \mathsf{a}'$ and $(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\neg\hat{\mathbb{F}}(cc)\rangle \curlywedge \mathsf{a}\ (\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \mathsf{a}''$ it follows that $(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\neg\hat{\mathbb{F}}(cc)\rangle \curlywedge \mathsf{a}'\ (\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \mathsf{a}''$;

2. from $\Psi \vdash \{\mathsf{a}''\}\,\mathbb{I}'$ and the induction hypodissertation it follows that $\Psi' \vdash \{\mathsf{a}''\}\,\mathbb{I}'$;

3. from $\mathsf{a} \Rightarrow \mathsf{a}'$ and $(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\hat{\mathbb{F}}(cc)\rangle \curlywedge \mathsf{a}\ (\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \Psi(\mathtt{f})$ it follows that $(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\hat{\mathbb{F}}(cc)\rangle \curlywedge \mathsf{a}'\ (\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \Psi(\mathtt{f})$.

Case JMPR. The derivation $\mathcal{D}$ has the form

$$\frac{\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}'') \qquad \mathsf{a} \Rightarrow \mathsf{a}''}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{jmp}\ \mathtt{r}}$$

From $\mathsf{a} \Rightarrow \mathsf{a}'$, $\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')$, and $\mathsf{a} \Rightarrow \mathsf{a}''$ it follows that $\mathsf{a}' \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')$, and $\mathsf{a}' \Rightarrow \mathsf{a}''$.

Case ECP. The derivation $\mathcal{D}$ has the form

$$\frac{(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f})) \curlywedge \mathsf{a}\ \mathbb{S}) \Rightarrow \mathsf{a}'' \qquad \mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{\mathsf{a}''\}\,\mathbb{I}}{\Psi \vdash \{\mathsf{a}\}\,\mathbb{I}}$$

We have

1. from $\mathsf{a} \Rightarrow \mathsf{a}'$ and $(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f})) \curlywedge \mathsf{a}\ \mathbb{S}) \Rightarrow \mathsf{a}''$ it follows that $(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi(\mathtt{f})) \curlywedge \mathsf{a}'\ \mathbb{S}) \Rightarrow \mathsf{a}''$;

2. from $\Psi \vdash \{\mathsf{a}''\}\,\mathbb{I}$ by the induction hypodissertation it follows that $\Psi' \vdash \{\mathsf{a}''\}\,\mathbb{I}$.

Case CALLI. The derivation $\mathcal{D}$ has the form

$$\frac{\mathsf{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{call}\ \mathtt{f};[\mathtt{f}_{ret}]}$$

From $\mathsf{a} \Rightarrow \mathsf{a}'$ and $\mathsf{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \Psi(\mathtt{f})$, it follows that $\mathsf{a}' \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \Psi(\mathtt{f})$.

Case CALLR. The derivation $\mathcal{D}$ has the form

$$\frac{\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}'') \qquad \mathsf{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \mathsf{a}''}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{call}\ \mathtt{r};[\mathtt{f}_{ret}]}$$

From $\mathsf{a} \Rightarrow \mathsf{a}'$, $\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}'')$, and $\mathsf{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \mathsf{a}''$, it follows that $\mathsf{a}' \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}'')$, and $\mathsf{a}' \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \mathsf{a}''$.

Case RET. The derivation $\mathcal{D}$ has the form

$$\frac{\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}'') \qquad \mathsf{a} \Rightarrow_{\mathsf{pop}} \mathsf{a}''}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{ret}}$$

From $\mathsf{a} \Rightarrow \mathsf{a}'$, $\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}'')$, and $\mathsf{a} \Rightarrow_{\mathsf{pop}} \mathsf{a}''$, it follows that $\mathsf{a}' \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}'')$, and $\mathsf{a}' \Rightarrow_{\mathsf{pop}} \mathsf{a}''$. ■

Lemma D.2 (XCAP86 Code Heap Typing)

If $\Psi_{IN} \vdash \mathbb{C}:\Psi$ and $\mathtt{f} \in \mathsf{dom}(\Psi)$ then $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

Proof Sketch. The proof is same as the XCAP Code Heap Typing (Lemma B.2). ■

Lemma D.3 (XCAP86 State Typing)

If $\Psi_{IN} \vdash \mathbb{C}:\Psi$ and $[\![\mathsf{cptr}(\mathtt{f},\mathsf{a})]\!]_{\Psi}$ then $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi_{IN} \vdash \{\mathsf{a}\}\,\mathbb{C}(\mathtt{f})$.

Proof Sketch. The proof is same as the XCAP State Typing (Lemma B.3). ■

Lemma D.4 (XCAP86 Progress)

If $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Proof Sketch. Derivation $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$ has the following form.

$$\frac{\Psi_G \vdash \mathbb{C}:\Psi_G \qquad ((\mathsf{DC}(\mathbb{C}) * [\![\mathsf{a}]\!]_{\Psi_G})\ \mathbb{S}) \qquad \mathsf{lookup}(\mathbb{C},pc,\mathbb{I}) \qquad \Psi_G \vdash \{\mathsf{a}\}\,\mathbb{I}}{\Psi_G \vdash \{\mathsf{a}\}\,(\mathbb{S},pc)}$$

The proof is by induction over derivation $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{I}$.

Case SEQ, JMPI, JCC, JMPR, CALLI, CALLR, and RET. the proof is by simple inspections.

Case ECP. The proof is by the induction hypodissertation. ■

Lemma D.5 (XCAP86 Preservation)

If $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$ then there exists an assertion a' such that $\Psi_G \vdash \{\mathsf{a}'\}\,\mathbb{P}'$.

Proof Sketch. Suppose $\mathbb{P} = ((\mathbb{H},\mathbb{R},\mathbb{F}),pc)$. We name derivation $\Psi_G \vdash \{\mathsf{a}\}\,\mathbb{P}$ as $\mathcal{D}$, which has the following form.

$$\frac{\Psi_G \vdash \mathbb{C}:\Psi_G \qquad ((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F})) \qquad \mathsf{lookup}(\mathbb{C},pc,\mathbb{I}) \qquad \Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi_G \vdash \{\mathtt{a}\}\,((\mathbb{H},\mathbb{R},\mathbb{F}),pc)}$$

The proof is by induction over the derivation $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}$, named as $\mathcal{E}$.

Case SEQ. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' \qquad \Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}'}{\Psi_G \vdash \{\mathtt{a}\}\,\mathtt{c};\mathbb{I}'}$$

By the operational semantics, $\mathbb{P}' = (\mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R},\mathbb{F}),npc)$. Then

1. $\Psi_G \vdash \mathbb{C}:\Psi_G$ is in $\mathcal{D}$;

2. from $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$ and $\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}'$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}']\!]_{\Psi_G})\ \ \mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R},\mathbb{F}))$;

3. from $\mathsf{lookup}(\mathbb{C},pc,\mathbb{I})$ and $\mathsf{Dc}(\mathbb{H},pc) = (\mathtt{c},npc)$ it follows $\mathsf{lookup}(\mathbb{C},npc,\mathbb{I}')$;

4. $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}'$ is in $\mathcal{E}$.

Case JMPI. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow \Psi_G(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi_G)}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{f}}$$

By the operational semantics, $\mathbb{P}' = ((\mathbb{H},\mathbb{R},\mathbb{F}),\mathtt{f})$. Then

1. $\Psi_G \vdash \mathbb{C}:\Psi_G$ is in $\mathcal{D}$;

2. from $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$ and $\mathtt{a} \Rightarrow \Psi_G(\mathtt{f})$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\Psi_G(\mathtt{f})]\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$;

3. from $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ and $\Psi_G \vdash \mathbb{C}:\Psi_G$ by Code Heap Typing (Lemma D.2) it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, and it follows that $\mathsf{lookup}(\mathbb{C},\mathtt{f},\mathbb{C}(\mathtt{f}))$;

4. from $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ and $\Psi_G \vdash \mathbb{C}:\Psi_G$ by Code Heap Typing (Lemma D.2) it follows that $\Psi_G \vdash \{\Psi_G(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

Case JCC. The derivation $\mathcal{E}$ has the form

$$\frac{\begin{array}{c}(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\neg\hat{\mathbb{F}}(cc)\rangle \;\mathbin{\wedge\!\!\!\wedge}\; \mathtt{a}\;(\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \mathtt{a}' \qquad \Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}' \\ (\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\hat{\mathbb{F}}(cc)\rangle \;\mathbin{\wedge\!\!\!\wedge}\; \mathtt{a}\;(\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \Psi_G(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi_G)\end{array}}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{j}cc\;\mathtt{f};\mathbb{I}'}$$

By the operational semantics, when $\hat{\mathbb{F}}(cc)$, $\mathbb{P}' = ((\mathbb{H},\mathbb{R},\mathbb{F}),\mathtt{f})$. Then

1. $\Psi_G \vdash \mathbb{C}\!:\!\Psi_G$ is in $\mathcal{D}$;

2. from $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\;(\mathbb{H},\mathbb{R},\mathbb{F}))$ and $(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\hat{\mathbb{F}}(cc)\rangle \;\mathbin{\wedge\!\!\!\wedge}\; \mathtt{a}\;(\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \Psi_G(\mathtt{f})$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\Psi_G(\mathtt{f})]\!]_{\Psi_G})\;(\mathbb{H},\mathbb{R},\mathbb{F}))$;

3. from $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ and $\Psi_G \vdash \mathbb{C}\!:\!\Psi_G$ by Code Heap Typing (Lemma D.2) it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, and it follows that $\mathsf{lookup}(\mathbb{C},\mathtt{f},\mathbb{C}(\mathtt{f}))$;

4. from $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ and $\Psi_G \vdash \mathbb{C}\!:\!\Psi_G$ by Code Heap Typing (Lemma D.2) it follows that $\Psi_G \vdash \{\Psi_G(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

By the operational semantics, when $\neg\hat{\mathbb{F}}(cc)$, $\mathbb{P}' = ((\mathbb{H},\mathbb{R},\mathbb{F}),npc)$. Then

1. $\Psi_G \vdash \mathbb{C}\!:\!\Psi_G$ is in $\mathcal{D}$;

2. from $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\;(\mathbb{H},\mathbb{R},\mathbb{F}))$ and $(\lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\langle\neg\hat{\mathbb{F}}(cc)\rangle \;\mathbin{\wedge\!\!\!\wedge}\; \mathtt{a}\;(\mathbb{H},\mathbb{R},\mathbb{F})) \Rightarrow \mathtt{a}'$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}']\!]_{\Psi_G})\;(\mathbb{H},\mathbb{R},\mathbb{F}))$;

3. from $\mathsf{lookup}(\mathbb{C},pc,\mathbb{I})$ and $\mathsf{Dc}(\mathbb{H},pc) = (\mathtt{c},npc)$ it follows $\mathsf{lookup}(\mathbb{C},npc,\mathbb{I}')$;

4. $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}'$ is in $\mathcal{E}$.

Case JMPR. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\;\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}') \qquad \mathtt{a} \Rightarrow \mathtt{a}'}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{jmp}\;\mathtt{r}}$$

From $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\;(\mathbb{H},\mathbb{R},\mathbb{F}))$, $\mathtt{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\;\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}')$ and $\mathtt{a} \Rightarrow \mathtt{a}'$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}']\!]_{\Psi_G})\;(\mathbb{H},\mathbb{R},\mathbb{F}))$ and $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}')]\!]_{\Psi_G}$.
By the operational semantics, $\mathbb{P}' = ((\mathbb{H},\mathbb{R},\mathbb{F}),\mathbb{R}(\mathtt{r}))$. Then

1. $\Psi_G \vdash \mathbb{C}\!:\!\Psi_G$ is in $\mathcal{D}$;

2. $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}']\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$ is from above;

3. by $\Psi_G \vdash \mathbb{C}{:}\Psi_G$, $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}')]\!]_{\Psi_G}$, and State Typing (Lemma D.3) it follows that $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\mathbb{C})$, and it follows that $\mathsf{lookup}(\mathbb{C},\mathbb{R}(\mathtt{r}),\mathbb{C}(\mathbb{R}(\mathtt{r})))$;

4. by $\Psi_G \vdash \mathbb{C}{:}\Psi_G$, $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}')]\!]_{\Psi_G}$, and State Typing (Lemma D.3) it follows that $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{R}(\mathtt{r})$.

Case ECP. The derivation $\mathcal{E}$ has the form

$$\frac{(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi_G(\mathtt{f})) \mathrel{\wedge\!\!\!\wedge} \mathtt{a}\ \mathbb{S}) \Rightarrow \mathtt{a}' \qquad \mathtt{f} \in \mathsf{dom}(\Psi_G) \qquad \Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}}{\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I}}$$

From $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$, $(\lambda\mathbb{S}.\,\mathsf{cptr}(\mathtt{f},\Psi_G(\mathtt{f})) \mathrel{\wedge\!\!\!\wedge} \mathtt{a}\ \mathbb{S}) \Rightarrow \mathtt{a}'$, and $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}']\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$. Together with $\Psi_G \vdash \{\mathtt{a}'\}\,\mathbb{I}$, the prove is completed by using the induction hypodissertation.

Case CALLI. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \Psi_G(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi_G)}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{call}\ \mathtt{f};[\mathtt{f}_{ret}]}$$

By the operational semantics, $\mathbb{P}' = (\mathtt{Next}_{\mathsf{push}\ \mathtt{f}_{ret}}(\mathbb{H},\mathbb{R},\mathbb{F}),\mathtt{f})$. Then

1. $\Psi_G \vdash \mathbb{C}{:}\Psi_G$ is in $\mathcal{D}$;

2. from $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\ \ (\mathbb{H},\mathbb{R},\mathbb{F}))$ and $\mathtt{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \Psi_G(\mathtt{f})$ it follows that $((\mathsf{DC}(\mathbb{C}) * [\![\Psi_G(\mathtt{f})]\!]_{\Psi_G})\ \ \mathtt{Next}_{\mathsf{push}\ \mathtt{f}_{ret}}(\mathbb{H},\mathbb{R},\mathbb{F}))$;

3. from $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ and $\Psi_G \vdash \mathbb{C}{:}\Psi_G$ by Code Heap Typing (Lemma D.2) it follows that $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$, and it follows that $\mathsf{lookup}(\mathbb{C},\mathtt{f},\mathbb{C}(\mathtt{f}))$;

4. from $\mathtt{f} \in \mathsf{dom}(\Psi_G)$ and $\Psi_G \vdash \mathbb{C}{:}\Psi_G$ by Code Heap Typing (Lemma D.2) it follows that $\Psi_G \vdash \{\Psi_G(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})$.

Case CALLR. The derivation $\mathcal{E}$ has the form

$$\frac{\mathtt{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}') \qquad \mathtt{a} \Rightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \mathtt{a}'}{\Psi_G \vdash \{\mathtt{a}\}\,\mathsf{call}\ \mathtt{r};[\mathtt{f}_{ret}]}$$

From $((\mathsf{DC}(\mathbb{C}) * [\![\mathtt{a}]\!]_{\Psi_G})\ (\mathbb{H},\mathbb{R},\mathbb{F}))$, $\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')$ and $\mathsf{a} \Rrightarrow_{\mathsf{push}\ \mathtt{f}_{ret}} \mathsf{a}'$ it follows that $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')]\!]_{\Psi_G}$ and $((\mathsf{DC}(\mathbb{C}) * [\![\mathsf{a}']\!]_{\Psi_G})\ \mathtt{Next}_{\mathsf{push}\ \mathtt{f}_{ret}}(\mathbb{H},\mathbb{R},\mathbb{F}))$.

By the operational semantics, $\mathbb{P}' = (\mathtt{Next}_{\mathsf{push}\ \mathtt{f}_{ret}}(\mathbb{H},\mathbb{R},\mathbb{F}),\mathbb{R}(\mathtt{r}))$. Then

1. $\Psi_G \vdash \mathbb{C}{:}\Psi_G$ is in $\mathcal{D}$;
2. $((\mathsf{DC}(\mathbb{C}) * [\![\mathsf{a}']\!]_{\Psi_G})\ \mathtt{Next}_{\mathsf{push}\ \mathtt{f}_{ret}}(\mathbb{H},\mathbb{R},\mathbb{F}))$ is from above;
3. by $\Psi_G \vdash \mathbb{C}{:}\Psi_G$, $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')]\!]_{\Psi_G}$, and State Typing (Lemma D.3) it follows that $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\mathbb{C})$, and it follows that $\mathsf{lookup}(\mathbb{C},\mathbb{R}(\mathtt{r}),\mathbb{C}(\mathbb{R}(\mathtt{r})))$;
4. by $\Psi_G \vdash \mathbb{C}{:}\Psi_G$, $[\![\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathsf{a}')]\!]_{\Psi_G}$, and State Typing (Lemma D.3) it follows that $\Psi_G \vdash \{\mathsf{a}'\}\,\mathbb{C}(\mathbb{R}(\mathtt{r}))$.

Case RET. The derivation $\mathcal{E}$ has the form

$$\frac{\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}') \qquad \mathsf{a} \Rrightarrow_{\mathsf{pop}} \mathsf{a}'}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{ret}}$$

From $((\mathsf{DC}(\mathbb{C}) * [\![\mathsf{a}]\!]_{\Psi_G})\ (\mathbb{H},\mathbb{R},\mathbb{F}))$, $\mathsf{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}')$ and $\mathsf{a} \Rrightarrow_{\mathsf{pop}} \mathsf{a}'$ it follows that $[\![\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}')]\!]_{\Psi_G}$ and $((\mathsf{DC}(\mathbb{C}) * [\![\mathsf{a}']\!]_{\Psi_G})\ \mathtt{Next}_{\mathsf{pop}}(\mathbb{H},\mathbb{R},\mathbb{F}))$.

By the operational semantics, $\mathbb{P}' = (\mathtt{Next}_{\mathsf{pop}}(\mathbb{H},\mathbb{R},\mathbb{F}),\mathbb{H}(\mathbb{R}(\mathsf{esp})))$. Then

1. $\Psi_G \vdash \mathbb{C}{:}\Psi_G$ is in $\mathcal{D}$;
2. $((\mathsf{DC}(\mathbb{C}) * [\![\mathsf{a}']\!]_{\Psi_G})\ (\mathbb{H},\mathbb{R},\mathbb{F}))$ is from above;
3. by $\Psi_G \vdash \mathbb{C}{:}\Psi_G$, $[\![\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}')]\!]_{\Psi_G}$, and State Typing (Lemma D.3) it follows that $\mathbb{H}(\mathbb{R}(\mathsf{esp})) \in \mathsf{dom}(\mathbb{C})$, and it follows that $\mathsf{lookup}(\mathbb{C},\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathbb{C}(\mathbb{H}(\mathbb{R}(\mathsf{esp}))))$;
4. by $\Psi_G \vdash \mathbb{C}{:}\Psi_G$, $[\![\mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathsf{a}')]\!]_{\Psi_G}$, and State Typing (Lemma D.3) it follows that $\Psi_G \vdash \{\mathsf{a}'\}\,\mathbb{C}(\mathbb{H}(\mathbb{R}(\mathsf{esp})))$. ■

Theorem D.6 (XCAP86 Soundness)

If $\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^{n} \mathbb{P}'$.

Proof Sketch. By simple induction on n and using Progress (Lemma D.4) and Perservation (Lemma D.5). ■

Bibliography

[1] A. Ahmed and D. Walker. The logical approach to stack typing. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 74–85. ACM Press, 2003.

[2] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] A. J. Ahmed. Mutable fields in a semantic model of types. Talk presented at 2000 PCC Workshop, June 2000.

[4] A. J. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *Proc. 18th IEEE Symposium on Logic in Computer Science*, pages 33–44, June 2003.

[5] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[6] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 243–253, Jan. 2000.

[7] A. W. Appel and T. Jim. Continuation-passing, closuring-passing style. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, USA, Jan. 1989. ACM Press.

[8] A. W. Appel and D. McAllester. An indexed model of recursive types for founda-

tional proof-carrying code. Technical Report CS-TR-629-00, Princeton University, Dept. of Computer Science, Nov. 2000. To appear in TOPLAS.

[9] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[10] A. W. Appel, P.-A. Mellies, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th ACM Symposium on Principles of Programming Languages*, Jan. 2007.

[11] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound tal for back-end optimization. In *Proc. 2003 ACM Conference on Programming Language Design and Implementation*, pages 208–219. ACM Press, 2003.

[12] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conference on Programming Language Design and Implementation*, pages 95–107, New York, 2000. ACM Press.

[13] K. Crary. Toward a foundational typed assembly language. In *Proc. 30th ACM Symposium on Principles of Programming Languages*, page 198, Jan. 2003.

[14] K. Crary and J. C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, pages 191–205, 2002.

[15] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[16] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM Conference on Programming Language Design and Implementation*, pages 59–69, New York, 2001. ACM Press.

[17] R. S. Engelschall. GNU Pth - the GNU portable threads. http://www.gnu.org/software/pth/, 1999-2003.

[18] A. Felty. Semantic models of types and machine instructions for proof-carrying code. Talk presented at 2000 PCC Workshop, June 2000.

[19] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. Workshop on Types in Language Design and Implementation*, Jan. 2007.

[20] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 International Conference on Functional Programming*, pages 254–267, Sept. 2005.

[21] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 401–414, New York, NY, USA, June 2006. ACM Press.

[22] R. W. Floyd. Assigning meaning to programs. *Communications of the ACM*, Oct. 1967.

[23] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher-Order Logics*, pages 2–16. Springer-Verlag, 2005.

[24] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 250–261, Jan. 1999.

[25] M. Gordon. A mechanized Hoare logic of state transitions. In A. W. Roscoe, editor, *A Classical Mind—Essays in Honour of C.A.R. Hoare*, pages 143–160. Prentice Hall, 1994.

[26] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[27] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[28] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on the Applications of Higher Order Logic Theorem Proving*, pages 118–135. Springer-Verlag, September 2004.

[29] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, July 2002.

[30] H. Herbelin, F. Kirchner, B. Monate, and J. Narboux. Faq about coq. `http://pauillac.inria.fr/coq/doc/faq.html#htoc38`.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.

[32] T. Hoare. The verifying compiler: A grand challenge for computing research. In *Proc. 2003 International Conference on Compiler Construction (CC'03), Lecture Notes in Computer Science, Volume 2622*, pages 262–272, Warsaw, Poland, Apr. 2003. Springer-Verlag Heidelberg.

[33] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, Oct. 2005.

[34] Intel Corporation. *Intel Architecture Software Developer's Manual*, volume 1-3. Intel Corporation, 1997.

[35] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[36] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Dept. of Computer Science, Yale University, New Haven, CT, Jan. 2002.

[37] Microsoft Corp., *et al.* Common language infrastructure. Drafts of the ECMA TC39/TG3 standardization process., 2001.

[38] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[39] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[40] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In X. Leroy and A. Ohori, editors, *Proc. 1998 International Workshop on Types in Compilation: LNCS Vol 1473*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.

[41] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[42] D. A. Naumann. Predicate transformer semantics of a higher-order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, 2001.

[43] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, New York, Jan. 1997. ACM Press.

[44] G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.

[45] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[46] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[47] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd Symp. on Principles of Prog. Lang.*, Jan. 2006.

[48] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. In *Bulletin of Symbolic Logic, Volume 5(2)*, pages 215–144, June 1999.

[49] P. W. O'Hearn and R. D. Tennent. *Algol-Like Languages*. Birkhauser, Boston, 1997.

[50] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[51] F. Pfenning. Automated theorem proving. `http://www-2.cs.cmu.edu/~fp/courses/atp/`, Apr. 2004.

[52] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. 1988 ACM Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[53] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[54] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Los Alamitos, California, 2002. IEEE Computer Society.

[55] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 217–232. ACM Press, Jan. 2002.

[56] G. Tan. *A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code*. PhD thesis, Princeton University, 2005.

[57] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. 1996 ACM Conference on Programming Language Design and Implementation*, pages 181–192. ACM Press, 1996.

[58] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.

[59] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[60] H. Xi and R. Harper. A dependently typed assembly language. In *Proc. 6th ACM SIGPLAN International Conference on Functional Programming*, pages 169–180. ACM Press, Sept. 2001.

[61] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

[62] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, April 2003.

[63] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, Mar. 2004.

[64] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 International Conference on Functional Programming*, Sept. 2004.

[65] Y. Yu. *Automated Proofs of Object Code For A Widely Used Microprocessor*. PhD thesis, The University of Texas at Austin, 1992.